

▲ 1 Motivation und Einführung

Streitfragen um die „richtige“ Architektur von Betriebssystemen und Rechnern bestimmen vielfach die (ver-) öffentliche Meinungsbildung und die Werbeaussagen großer Hersteller. Jenseits der plakativen und vermeindlichen Vergleiche und Entwicklungssprünge haben sich jedoch die der Rechnerarchitektur und dem Aufbau von Betriebssystem zugrundeliegenden Konzepte in den letzten Jahren nicht umwälzend geändert. Vielmehr existiert eine stabile Basis gut verstandener und in der Praxis erprobter Grundlagen, die auch in modernen Rechnerarchitekturen und Betriebssystemen Anwendung finden. Hierzu zählt der Bestand an Wissen über Aufbau und interne Organisation von Prozessoren und Betriebssystemen, hierbei insbesondere Aspekte der Ablaufsteuerung, Speicherverwaltung und des Zugriffs auf Hintergrundspeicher. Gleichzeitig eroberten sich verteilte Systeme in der jüngeren Vergangenheit einen solch zentralen Stellenwert in der Anwendung, daß Kenntnisse über deren Basismechanismen inzwischen auch als Grundlagenwissen anzusehen sind.

1.1 Rechner- und Betriebssystementwicklung

Betriebssysteme und die Architektur der sie ausführenden Hardware sind einem stetigen Wandel unterworfen und haben sich im Verlauf der Entwicklung elektronischer Datenverarbeitung stark verändert. Im Kern bilden beide zusammen Triebfeder und Schrittmacher der verschiedenen Generationen und wirkten mit ihren Konzepten vielfach stilbildend für die jeweils aktuelle Form der Datenverarbeitung.

Technisch gesehen ist ein Betriebssystem lediglich ein Programm, welches durch die zugrundeliegende Hardware zur Ausführung gebracht wird. Jedoch nimmt dieser Programmtyp eine solche zentrale Stellung ein, daß vielfach Hardware und Betriebssystem (immernoch) als Einheit betrachtet werden, obwohl sich inzwischen beide voneinander entkoppelt -- jedoch durch starke Wechselwirkungen geprägt -- entwickeln.

Aufgrund dieser zentralen Stellung definiert DIN 44300 den Begriff *Betriebssystem* daher als: „Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften dieser Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen“.

Nachfolgend wird ein kurzer Überblick von Hardware-, Rechner- und Betriebssystemgenerationen gegeben, welche die historische Entwicklung kurz umreißt und einen Einblick in die Fortschritte der zugrundeliegenden Konzepte bietet. Gleichzeitig finden typische Interaktionsformen mit der aus Hardware und Betriebssystem gebildeten Recheneinheit Berücksichtigung.

Zeitraum	Schaltungsrealisierung	Verbreitung	Typische Betriebssysteme	Typische Programmiersprachen	Geschwindigkeit	Beispiele
1945-1955	Relais, später Röhren	wenige Exemplare (Univac: 46 Stück)	Keine	Steckkarten, später Maschinencode	10 ³ Ops./Sec.	Univac
1955-1965	Transistoren und Dioden	Hunderte	Hardwarespezifisch (z.B. Multics,)	Assembler, später FORTRAN, Algol60 und COBOL	10 ⁴ Ops./Sec.	IBM 1401 , 7094 , DEC PDP
1965-1980	Integrierte Schaltungen	Tausende	Hardwarespezifisch (z.B. OS/360, ...)	höhere Programmiersprachen, erste Programmiermethoden	10 ⁶ Ops./Sec.	IBM 360
ab 1980	Hoch integrierte Schaltungen (VLSI)	Millionen	Prozessorspezifisch (z.B. MS-DOS, Windows, Minix, Linux)	Höhere Programmiersprachen (C ++ , Java, ...)	min. 10 ⁷ Ops./Sec.	Intel Prozessoren, AMD, Motorola ...

Handelte es sich bei der Univac noch um ein eher experimentelles und nur in wenigen Stückzahlen gefertigtes Gerät, die sich durch hohe Anschaffungs- und Betriebskosten sowie immensen Platzbedarf und geringe Zuverlässigkeit auszeichneten, so beanspruchen die die ab 1955 entwickelten Maschinen ihre Rolle als erste kommerziell erfolgreiche Systeme. Für sie hat sich die Gattungsbezeichnung *Mainframe* eingebürgert. Die Arbeit mit den Rechnern dieser Generation läuft ausschließlich offline im *Stapelverarbeitungsbetrieb* (auch: *batch-Betrieb*), d.h. die zu berechnenden Aufgaben (*jobs*) werden nicht interaktiv abgearbeitet, sondern nach ihrer Zeitdauer manuell durch einen menschlichen Operator zusammengestellt und der Maschine durch Lochkarten vorgelegt. Zumeist sind die Maschinen für genau ein konkretes Anwendungsgebiet (z.B. numerische Berechnungen) konzipiert.

Später entwickelt sich daraus die *Stapelverarbeitung*, welche die Eingaben in Form von Magnetbändern -- jedoch immernoch seriell und nicht interaktiv -- akzeptiert.

Ab 1965 ist der Versuch einer Konsolidierung der verschiedenen Hardwarestränge seitens der Hersteller zu spüren. So fertigt IBM mit dem System 360 ein Serie kompatibler Rechner (d.h. die auf einer Systemvariante ablauffähige Software kann auch auf einem anderen Mitglied der Rechnerfamilie zur Ausführung gebracht werden). In dieser Entwicklungsgeneration steht bereits sowohl Rechenkapazität zur Verfügung, daß ein aktuell bearbeiteter Auftrag die Maschinen nicht mehr vollständig auszulasten vermag. Daher wird mit dem Konzept des *Multiprogramming* die Möglichkeit zur Bearbeitung einer anderen anstehenden Aufgabe geschaffen, während sich die aktuell in Bearbeitung befindliche im Wartezustand (etwa auf Ende einer Ein-/Ausgabeoperation) befindet. Diese Ausführungsform bedingt jedoch eine zusätzliche Komplexität im Aufbau der Verwaltungsfunktionen, da diese nun die gleichzeitige Präsenz verschiedener Programme im verwalteten Speicher zu berücksichtigen haben. Darüberhinaus ist der Programmablauf so zu organisieren, daß sich die verschiedenen Programme geschützt voneinander abgearbeitet werden und sich nicht

gegenseitig beeinträchtigen.

In Erweiterung des Multiprogrammbetriebes wurde mit *Timesharing* eine Variante dieser Betriebsform eingeführt, die es erstmals mehreren Benutzern gestattete gleichzeitig online am System zu Arbeiten. Durch schnelles Umschalten zwischen den Aufgaben der Einzelnutzer entsteht so die Illusion der alleinigen Arbeit am System.

Mit wachsenden Hauptspeichergrößen führt die dritte Entwicklungsgeneration (die sog. *Minicomputer*) auch Techniken zur Lösung der Abhängigkeit von langsamen Speichertypen ein. So wurde mit *Simultaneous Peripheral Operation On Line* -- kurz *Spooling* -- ein Verfahren eingeführt alle übergebenen Aufgaben zunächst von Lochkarten einzulesen und auf Platten zwischenspeichern.

Aus einer Weiterentwicklung des Timesharing-Systems *MULTIC* (MULTIplexed Information and Computing System) entwickelt sich ab den frühen 1970er Jahren das System *UNIX*, welches zunächst auf der PDP-7-Hardware entwickelt, jedoch später auch für andere Plattformen portiert, wurde.

Die Hardware- und Betriebssystementwicklung ab 1980 ist begleitet durch einen massiven Siegeszug an einem immer größer werdenden Massenmarkt. Die Fortschritte der Fertigungstechnik erlauben eine immer weiter voranschreitende Packungsdichte der integrierten Schaltungen und verbilligen gleichzeitig deren Herstellung. Die so gefertigten Hardwaren (als *Personal Computer* (PC) oder auch *Mikrocomputer* bezeichnet) entsprechen in ihren Strukturprinzipien weitestgehend den Minicomputern der Vorgängergeneration, jedoch zu einem sehr viel niedrigeren Verkaufspreis.

Als eines der ersten Betriebssysteme für die neue Rechnergeneration stand *CP/M* (*Control Program for Microcomputers*) zur Verfügung, welches den 1974 durch [Intel](#) vorgestellten 8-Bit Mikroprozessor **8080** unterstützte. Seit den frühen 1980er Jahren finden Prozessoren dieses Typs Verwendung in der, zunächst ausschließlich durch [IBM](#) vorangetriebenen, PC-Line. Nachdem Verhandlungen zwischen den Autoren des *CP/M-Systems* und IBM über die Nutzung des Betriebssystems für den *IBM PC* scheiterten lieferte [Microsoft](#) mit *MS-DOS* (*Microsoft Disk Operating System*) das erste kommerziell vertriebene Betriebssystem für diesen Rechner Typ.

Gegen Ende der 1980er Jahre geriet die Benutzbarkeit der -- damals schon in hohen Stückzahlen verkauften PCs -- immer stärker ins Zentrum des Interesses. Als Vorreiter führte der Hersteller Apple auf seiner nicht IBM-kompatiblen Hardware eine [graphische Oberfläche](#) (GUI -- *Graphical User Interface*) zur Bedienung des Rechners ein. Deren, ursprünglich am [Xerox PARC-Forschungszentrum](#) entwickeltes Konzept, gewann in der Folge immer größere Bedeutung und wurde letztlich durch das MS-DOS-Programm *Windows* popularisiert.

Seit 1995 vertreibt Microsoft ein -- ebenfalls *Windows* genannte -- Familie eigenständiger PC-Betriebssysteme, die den kommerziellen Massenmarkt beherrscht.

Gleichzeitig findet vielfach -- insbesondere auf Netzwerkrechnern (Servern) -- die ursprünglich durch den finnischen Studenten [Linus Torvalds](#) entwickelte -- Unix-Variante *Linux* breiten Einsatz.

Hinzu tritt in den 1990er Jahren der Trend zur verteilten Verarbeitung auf Basis durch Netzwerke gekoppelter Rechner.

1.2 Peripherie

Parallel zur Entwicklung der Kernhardware und der darauf ablaufenden Betriebssysteme entwickelt sich die Landschaft der mit der Berechnungseinheit kommunizierenden Peripheriegeräte. Hierbei ist der Begriff des Peripheriegerätes jedoch aus Sicht des Zentralprozessors interpretiert und daher entsprechend weiter gefasst als seine landläufige Definition, welche ausschließlich an der Zentraleinheit (bestehend aus CPU, Speicher und Bussen) anschließbare Geräte berücksichtigt.

Die erste Rechnergeneration verfügte noch über keine dedizierten Peripheriegeräte. Eingaben wurden direkt, durch Schalttafeln und Erstellung von Hardwareverbindungen, an der Maschine vorgenommen; Ausgaben ebendort -- von Lampen oder anderen Signalgebern -- abgelesen.

Ab den 1950er Jahren finden sich zunehmend externe Geräte zur Abwicklung und dauerhaften Speicherung der verarbeiteten Daten. Urvater des Ein- und Ausgabegerätes waren zunächst Lochstreifen, welche zur Steuerung automatisierter Webstühle bereits im 19. Jahrhundert und später zur Darstellung von Morsezeichen Verwendung fanden, die später durch Lochkarten abgelöst wurden.

Die Lochkarte bot, neben der vergleichsweise einfachen manuellen Handhabung die Möglichkeit bestehende -- zuvor durch Tabelliermaschinen verarbeitete -- Datenbestände weiter einsetzen zu können.

Im Deutschen war damals der Begriff *Hollerithmaschine* -- nach [H. Hollerith](#), der bereits zur Abwicklung der US-Volkszählung von 1890 Lochkarten einsetzte -- für lochkartenbetriebene Rechner gängig.

Zur Senkung der Fehlerrate (etwa durch Vertauschung zweier Lochkarten) und Erhöhung der Verarbeitungsgeschwindigkeit wurden die Lochkarten zunehmend durch Magnetbänder ersetzt.

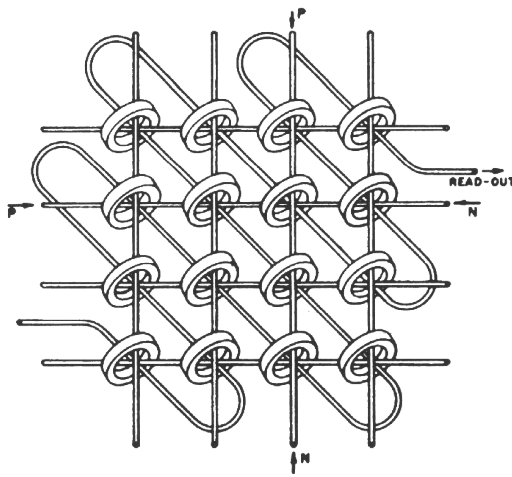
Das Aufkommen von Ferritkernspeichern markiert den Übergang zu Speicherorganisationsformen mit wahlfreiem Zugriff, da jeder einzelne Speicherring (Bit) separat angesprochen werden konnte.

Die Grundidee der Ferritkernspeicher (auch als *Ringkernspeicher* bekannt) hat sich [bis heute](#) erhalten, wenngleich ihre technische Realisierung im Lauf der Zeit natürlich mit den modernisierten Fertigungsmethoden Schritt hielt.

In einem Ringkernspeicher sind parallel zum Rahmen dünne Metalldrähte -- die Steuerdrähte -- gespannt, die mit Strom beschickt werden können. Zusätzlich verläuft ein Diagonaldraht (Lesedraht), der alle magnetisierbaren Ringkerne verbindet.

Jeder Ring kann separat durch gerichteten Stromfluß in den Steuerdrähten auf zwei verschiedene Weisen magnetisiert werden. Fließt der horizontale Steuerstrom transversal (in der Abbildung in *P*-Richtung) und der vertikale nach unten (ebenfalls *P*), so entsteht eine Magnetisierung die diagonal nach unten gerichtet ist. Im umgekehrten Falle (Stromrichtungen: aufwärts und linksgerichteter Strom (jeweils durch *N*) entsteht ein entgegengesetztes, d.h. diagonal nach links-oben gerichtetes Magnetfeld.

Abbildung 1: Aufbau eines Ringkernspeichers



(click on image to enlarge!)

Quelle: J. A. Rajchmann: A Myriabit Magnetic Core Matrix Memory. IRE Proceedings 1408, Okt. 1953, IEEE 1953.

Zum Auslesen des Speicherzustandes wird die Hystereseeigenschaft ausgenutzt und testweise ein beliebiger Beschreibungsvorgang durchgeführt. Ändert sich durch diesen die zuvor präsente Magnetisierung, so wird im Lesedraht ein Stromstoß induziert, andernfalls nicht.

Als Weiterentwicklung der Ringkernspeicher bürgerten sich *Trommelspeicher* ein, welche die Datenspeicherung auf einer magnetisierten rotierenden Trommel vornimmt. Das Auslesen geschieht durch am Gehäuse fest angebrachte Magnetköpfe.

Seit den 1960er Jahren (erster „Masseneinsatz“ in IBM 305) kommen zunehmend magnetische Speicherplatten mit wahlfreiem als externe Speicher in den Masseneinsatz. Ihre technischen Basiskonzepte unterscheiden sich kaum von aktuellen Festplatten.

Mit dem Aufkommen des interaktiven Timesharing-Betriebes wurden auch neue Formen der Ein- und Ausgabeverarbeitung notwendig, da nicht mehr jeder Nutzer direkten Zugriff auf das physische Speichermedium haben konnte. Daher entstanden visuelle Endgeräte zur Präsentation der Daten, sowie Tastaturen zu ihrer Erfassung.

Der Siegeszug der graphischen Benutzeroberflächen in den 1980er Jahren führte die *Maus* als neues Eingabemedium als festen Bestandteil der Rechnerperipherie ein.

Aktuelle Trends schließen Sprachein- und Ausgabe, sowie tastaturbasierte Eingabeformen mit reduzierter Tastenzahl (etwa auf Mobiltelefonen und PDAs) ein.

1.3 Zentralprozessoren

Wesentliches Element einer Rechnerarchitektur ist der *Zentralprozessor* (engl. *central processing unit*, kurz: CPU), welcher die Hauptteile der Verarbeitung übernimmt. Die CPU, deren stetiger Leistungszuwachs sich in der Vergangenheit nach dem *Moore'schen Gesetz* (es sagt auf Basis empirischer Daten eine Verdopplung der Transistorenanzahl pro Chip (und damit der Leistungsfähigkeit) alle 18 Monate voraus) entwickelte, ist in jüngerer Zeit selbst zum Marketingbestandteil avanciert.

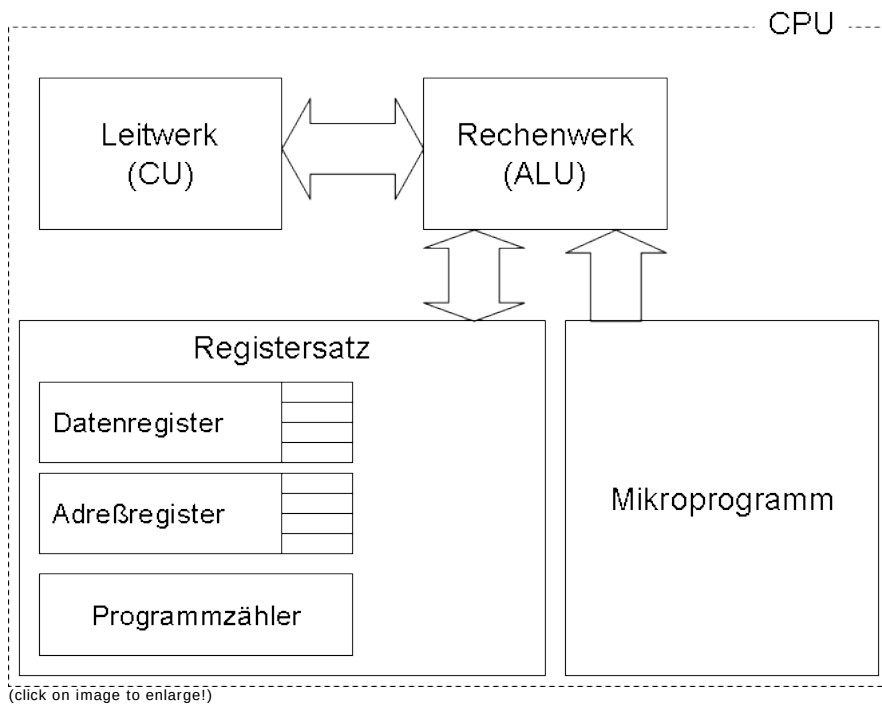
Gleichzeitig rücken die internen Realisierungsdetails stark in das Bewußtsein des Anwenders, da sie die Leistungsfähigkeit und wirtschaftlich einer gegebenen Hardware stark beeinflussen.

Historisch gesehen determiniert die verwendete CPU typischerweise die auf einer Hardware ausführbaren Programme. So sind übersetzte Anwendungen im Normalfall nur auf einem bestimmten Prozessortyp ausführbar, außer die verfügbare Hardware oder eine darauf angebotene Softwarekomponente bieten eine simulierende Unterstützung der ursprünglichen Zielhardware an. Unter Verwendung dieser simulierenden Unterstützung verhält sich ein Programm, welches für einen anderen Prozessortyp übersetzt wurde unter Eingabe derselben Daten so als würde es auf der ursprünglichen Hardware ausgeführt werden. Es handelt sich mithin um eine programmgesteuerte Imitation des Originalprozessors.

Diese imitierende Simulation kann an Leistungsfähigkeit die ursprüngliche Zielhardware durchaus signifikant übertreffen. Aus diesem Grunde hat IBM für das System 360 den Begriff der *Emulation* (wohl auch um den negativen Beigeschmack der Begriffe *Simulation* und *Imitat* zu vermeiden) geprägt, deren Prozessor Programme für den damals kommerziell erfolgreichen Großrechner IBM 7070 ausführen konnte. ([Cer03, S. 188])

Aktuelle Prozessoren folgen dem in [Abbildung 5](#) dargestellten schematischen Aufbau, welche die zentralen Architekturkomponenten *Leitwerk* (engl. *control unit* (CU)), *Rechenwerk* (engl. *arithmetic and logic unit* (ALU)), Registersatz und typischerweise den Mikroprogrammspeicher umfaßt.

Abbildung 2: Schematischer Aufbau eines Zentralprozessors



(click on image to enlarge!)

Diese, nach dem Initiator als von-Neumann -Architektur, bezeichnete Organisationsweise prägt den Aufbau von Rechenanlagen seit den späten 1940er Jahren und ist im Kern bis heute gültig; wenngleich sich in der Gegenwart verschiedene Engpässe dieses Ansatzes zeigen.

Die Kerngedanken der von-Neumann-Architektur sind:

- **Einteilung** der Verarbeitungseinheit in Leit- und Rechenwerk, Hauptspeicher, Ein- und Ausgabeeinheiten sowie Verbindungen (Busse) dazwischen.
Dabei bilden Leit- und Rechenwerk die CPU.
- Die verarbeiteten Daten werden binär repräsentiert und taktgesteuert in Worten fester Bitlängen verarbeitet.
- Programme und Daten werden in einem gemeinsamen Hauptspeicher unsepariert und gekennzeichnet abgelegt. Der Hauptspeicher ist fortlaufend organisiert und wird durch eindeutige Adressen angesprochen.
- Die Verarbeitung von Programmen und Daten erfolgt sequentiell typischerweise in der Reihenfolge der abgespeicherten Programmbefehle. Verzweigungen, Schleifen und Sprungbefehle können Abweichungen von der sequentiellen Abarbeitungsreihenfolge bedingen.

Die Festlegungen der von-Neumann-Architektur haben sich zunehmend als Gründe verschiedener Leistungsengpässe erweisen, die durch partielle Aufweichungen dieser Architekturprinzipien behoben werden können.

Einige bekannte Engpässe der von-Neumann-Architektur:

- Gleichbehandlung von Daten und darauf operierenden Programmen.
Die Zugriffe auf die im Hauptspeicher identisch abgelegten Daten und Programme können sich gegenseitig behindern, da keine separaten Ein-/Ausgabekanäle für diese verarbeiteten Inhalte vorgesehen sind.
Zusätzlich eröffnet die nicht erfolgte Separierung von Daten und Programmen eine besondere Klasse von Bedrohungsszenarien. So können geeignet gestaltete Datenblöcke sog. Pufferüberläufe (engl. *buffer overflows*) provozieren, bei denen über Datenbereiche ausführbarer (Angriffs-)Code in den Programmbereich eingeschleust wird. Beiden Problemen kann durch Einführung getrennter Speicher und Busse für Daten und Programme (insbesondere im Umfeld schneller Zwischenspeicher sog. *Caches*) begegnet werden (Harvard-Architektur). Ein neuerer Ansatz zur Beibehaltung der Ablage im selben Speicherbereich wird durch Einführung einer expliziten Kennzeichnung nicht ausführbarer Datenbereiche im Hauptspeicher realisiert.
- Sequentielle Verarbeitung.
Die Leistungsfähigkeit einer Hardware lässt sich durch Schaffung von Möglichkeiten zur gleichzeitigen (parallelen) Verarbeitung einzelner Befehle steigern. Ansätze hierzu bilden:
 - [Instruction Level Parallelism](#)
 - Multithreading
 - Multiprozessorsysteme
- Festgelegte Befehlsabfolge.
Die Verarbeitungsreihenfolge der Einzelbefehle eines Programmes kann von der Reihenfolge ihres Auftretens im Programm abweichen. In Verbindung mit Parallelitätsansätzen lassen sich so zeitaufwendige Befehlsfolgen „im Voraus“ berechnen und auf diesem Wege die Verarbeitungszeit senken.
Voraussetzung hierfür ist der flexible Umgang mit den belegbaren Registern, d.h. im Idealfalle die Existenz mehrerer Registersätze.

Die in [Abbildung 5](#) zusammengestellten Kernkomponenten einer CPU erfüllen festgelegte Aufgaben. Ihre Arbeitsteilung bildete sich im Laufe der Entwicklungsgeschichte heraus und ist bis heute aktuellen Prozessorarchitekturen nahezu unverändert präsent.

Rechenwerk (ALU)

Das Rechenwerk führt die Rechenoperationen auf den speicherresidenten Operatoren durch.

Aktuelle CPU-Architekturen (wie Intel Pentium, AMD Athlon und Power PC) verfügen innerhalb einer CPU über mehrere Rechenwerke, insbesondere solche für Fest- und Gleitkommaoperationen.

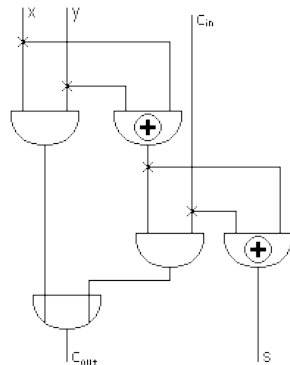
Typischerweise bietet ein Rechenwerk die folgenden grundlegenden Operationstypen an:

- Arithmetische Operationen: Addition, Subtraktion, Multiplikation und Division.
- Logische Operationen: AND, OR, NOT, NOR, XOR, ...
- Bitmanipulationen: Rotation, Verschiebung und Vergleiche.

Als Beispiel einer ALU-Komponente zur Addition n -stelliger Binärzahlen mit Übertrag sei der *Ripple-Carry-Addierer* betrachtet:

Grundkomponente eines solchen Addierers sind eine Reihe verschalteter Volladdierer (engl. *full adder* (FA)), wie er schematisch in [Abbildung 1](#) abgebildet ist.

Abbildung 3: Aufbau eines Volladdierers

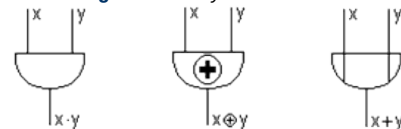


(click on image to enlarge!)

Abbildung aus: [Wikipedia, Volladdierer](#)

Anmerkung: Bedeutung der Schaltsymbole

Abbildung 4: Schaltsymbole



AND

XOR

OR

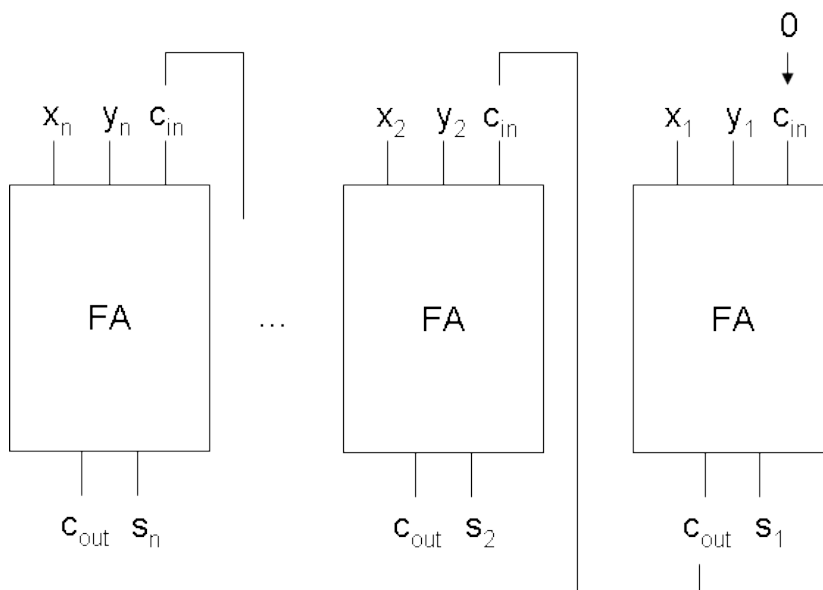
(click on image to enlarge!)

Im Unterschied zum Halbaddierer liefert der Volladdierer nicht nur einen Übertrag im Rahmen der Ergebnisberechnung, sondern akzeptiert diesen auch als Eingabe (einer möglicherweise vorhergehenden Additionsoperation).

Durch die Hintereinanderschaltung mehrerer Volladdierer und Verknüpfung des c_{in} -Einganges des n -ten Addierers mit dem c_{out} -Ausgang des vorhergehenden $n-1$ -ten Volladdierers entsteht die in [Abbildung 1](#) dargestellte Rechenwerkkomponente zur Berechnung n -stelliger Binärzahlen.

Zur Initialisierung der Übertragsbehandlung wird der Eingangsübertrag für den ersten Volladdierer mit 0 vorgelegt.

Abbildung 5: Ripple-Carry-Adder



(click on image to enlarge!)

Es ist keine zwingende Voraussetzung, daß alle Rechenoperationen, die von einer ALU bereitgestellt werden vollständig in Hardware realisiert sind. Durch die Verwendung von Mikroprogrammen können diese selbst durch

Programme implementiert sein.

Leitwerk (CU)

Dem Leitwerk obliegt die Steuerung des gesamten Verarbeitungsflusses. Hierzu zählen insbesondere die Versorgung der ALU mit Berechnungsaufgaben durch Bereitstellung der benötigten Befehle und Operanden.

Zur Regelung der Verarbeitung speichert das Leitwerk prozessorinterne Zustände (Status von Berechnungen (wie Überläufe, Ergebnis Null) und Fehlerindikatoren) in CPU-internen Speicherbereichen, den sog. Registern.

Registersatz

Der Registersatz wird durch eine Reihe von Speicherzellen, die direkt in der CPU untergebracht sind realisiert. Diese Speicherform stellt die schnellste (der Zugriff geschieht im Rahmen eines Prozessortaktzyklus) und gleichzeitig teuerste (daher stehen in der Regel in Summe nur wenige Byte zur Verfügung) dar.

Register besitzen typischerweise eine feste Bitlänge, die mit der verarbeitbaren Wortlänge der CPU in engem Zusammenhang steht. Üblich sind 8-, 16-, 32- und 64-Bit Register sowie 40- und 80-Bit Register für Spezialanwendungen (Fließkommaberechnung).

Häufig werden die zur Verfügung stehenden Register bezüglich ihrer Funktionsweise, d.h. ihres Einsatzgebietes unterschieden, da oftmals nicht in jedem Register Operanden für beliebige Maschineninstruktionen zur Verfügung gestellt werden können.

Minimalanforderung an einen Registersatz ist die Bereitstellung eines Befehlszeigerregisters (engl. *Instruction Pointer Register*), welches die Adresse des nächsten abzuarbeitenden Befehls enthält.

Darüberhinaus sind Register zur Aufnahme der Speicheradressen der zentralen Datenstrukturen (z.B. Stack) üblich.

Beispielsweise partitioniert die Intel-Pentium-Architektur ([IA-32](#)) den verfügbaren Registersatz in vier Bereiche:

- Allzweck (engl. *General-purpose*) Register:
Acht Register zur Aufnahme von Operanden und Speicheradressen.
 - EAX: Akkumulator zur Speicherung von Operanden und Ergebnisdaten.
 - EBX: Zeiger auf Daten im DS-Segment.
 - ECX: Zählregister für Schleifen und Zeichenkettenmanipulationen.
 - EDX: Zeiger auf Ein-/Ausgabedaten.
 - ESI: Zeiger in das durch DS bezeichnete Datensegment. Quellzeiger für Zeichenkettenoperationen.
 - EDI: Zeiger in das durch ES bezeichnete Datensegment zur Aufnahme von Zieldaten (typischerweise für Zeichenkettenoperationen verwendet).
 - ESP: Zeiger auf das oberste Element des Stacks, der durch das SS-Register bezeichnet wird.
 - EBP: Zeiger auf das unterste Element des Stacks, der durch das SS-Register bezeichnet wird.
- Segment Register:
Identifizieren Speicherbereiche.
 - DS, ES, FS, GS: Datensegmente, die Ein-/Ausgabedaten enthalten.
 - SS: Stack Segment.
- Status- und Kontrollregister:
Speichern Aussagen über verschiedene CPU interne Status.
 - CF (Carry Flag): Gesetzt falls bei einer arithmetischen Operation ein Über- oder Unterlauf auftritt.
 - PF (Parity Flag): Gesetzt falls das niedersignifikante Byte eines Resultats eine geradzahlige Anzahl 1-en enthält.
 - AF (Adjust Flag): Gesetzt falls bei einer Dezimalzahloperation ein Über- oder Unterlauf auftritt.
 - ZF (Zero Flag): Gesetzt falls das Ergebnis einer Berechnung Null ist.
 - SF (Sign Flag): Identisch zum höchstsignifikanten Bit eines Berechnungsergebnisses und damit identisch zum Vorzeichen.
 - OF (Overflow Flag): Gesetzt falls Berechnungsergebnis zu groß oder zu klein für Speicherung ist.
 - DF (Direction Flag): Steuert die Richtung der Stringverarbeitung.
 - IF (Interrupt Enable Flag): Steuert die Reaktion des Prozessors auf maskierbare Unterbrechungen.
 - TF (Trap Flag): Steuert den Eintritt in die Einzelschrittverarbeitung (zur Fehlersuche genutzt).
 - IOPL (IO Privilege Level): Bitfeld, welches die Ein-/Ausgabeprivilegien eines Prozesses festlegen.
 - NT (Nested Task Flag): Gesetzt falls aktuell verarbeitete Task mir vorhegend verarbeiteter in Verbindung steht.
 - RF (Resume Flag): Steuert Reaktion der CPU auf Debug-Ausnahmen.
 - VM (Virtual Mode): Steuert Eintritt in den virtuellen 8086-Modus.
 - AC (Alignment Flag): Aktiviert (in Verbindung mit dem AM-Bit des Kontrollregisters CR0) die Bereichsprüfung für Speicherreferenzen.
 - ID (Identification Flag): Weist auf Unterstützung des CPUID-Befehls hin.
- Befehlszeiger .
Das EIP-Register enthält einen 32-Bit langen Verweis auf die nächste auszuführende Instruktion.

Die durch eine CPU angebotenen Befehle können (wie im Falle des Ripple-Carry-Addierers gezeigt) festverdrahtet sein, oder selbst durch kleine in der CPU abgelegte Programme -- sog. *Mikroprogramme* -- realisiert sein.

Eine ausführliche Fallstudie zur IA-32- und IA-64-Architektur findet sich in [\[Mär01, S. 174ff.\]](#)

Mikroprogramm

Das Konzept der Mikroprogrammierung, d.h. Einzelbefehle in eine Reihe von Mikrooperationenaufzuteilen, wurde kommerziell erstmals im IBM-System 360 verwirklicht. Seine Wurzeln reichen jedoch noch in die Zeit der Vorgängermaschine 1620 zurück, die bereits durch geschickte Speicheroperationen eine Änderung des Befehlssatzes ermöglichte ([\[Cer03, S. 107\]](#)).

Die Existenz von Mikrooperationen ist ein zentrales Kennzeichen einer CISC-Architektur.

▲ 2 Rechnerarchitekturen

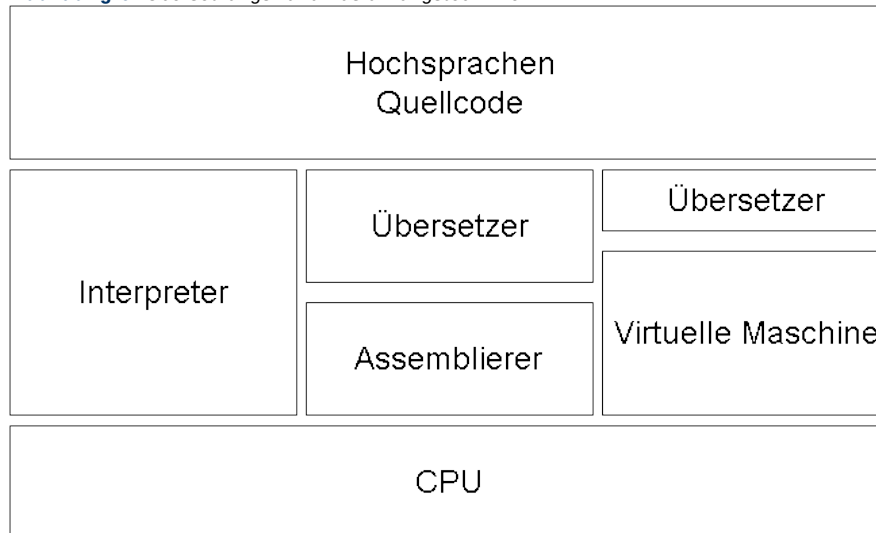
2.1 Befehlsarchitekturen

Die Architektur des Befehlssatzes, d.h. sein Aufbau und Umfang, bestimmt die Schnittstelle zur direkten Interaktion mit der CPU-Hardware welche typischerweise durch maschinell übersetzte (compilierte) Betriebssysteme und Anwendungsprogramme direkt bedient wird.

Die maschinelle Bedienung dieser Schnittstelle kann durch vielfältige Softwarekomponenten erfolgen.

[Abbildung 6](#) stellt drei in der Praxis bedeutsame Ansätze gegenüber.

Abbildung 6: Übersetzungs- und Ausführungstechniken



(click on image to enlarge!)

Im linken Bildbereich ist das Zusammenspiel einer ausschließlich interpretierenden Ausführungsumgebung dargestellt. Eine solche akzeptiert ein Quellprogramm als Eingabe und führt dieses „direkt“, d.h. ohne expliziten Übersetzerlauf, aus. Die notwendige Transformation der Hochspracheninstruktionen in die von der CPU unterstützten Befehle wird dynamisch zur Laufzeit vorgenommen.

Der mittlere Bereich stellt die klassischen Entwicklungsschritte unter Verwendung eines CPU-spezifischen (d.h. plattformspezifischen) Übersetzers dar, der maschinenspezifischen Assemblercode erzeugt, der vermöge eines Assemblers in direkt (nativ) ausführbaren Binärcode übersetzt wird.

Als Beispiel seien die im Verlauf der Übersetzung eines C-Programmes entstehenden Artefakte angeführt.

Quellcode:

```
#include <stdio.h>
int main(int argc, char** argv) {
    printf("hello world");
    return 0;
}
```

Erzeugter Assembler-Code für die Intel-Architektur

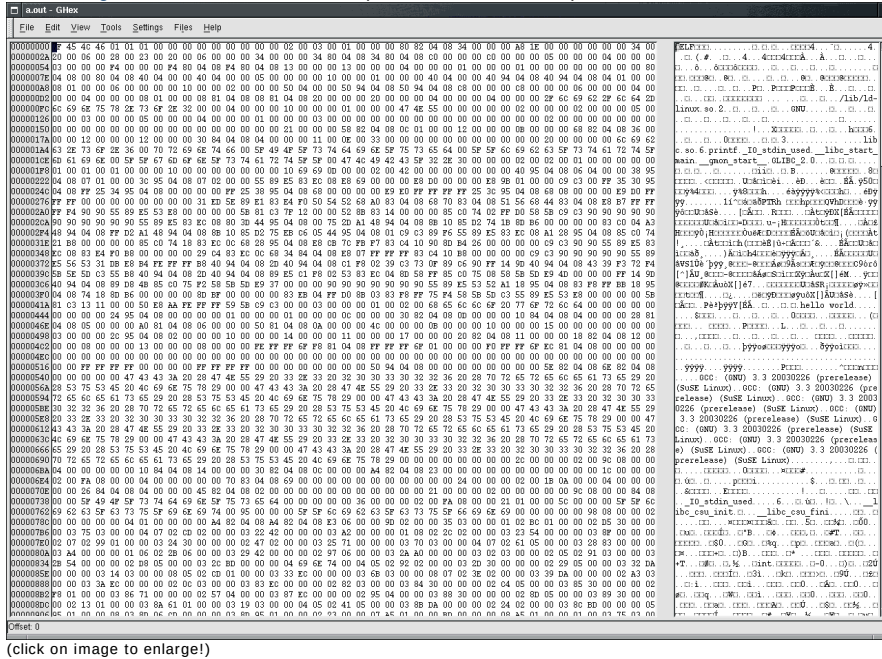
```
section .data
hello:    db 'hello world',10
helloLen: equ $-hello
section .text
global _start
_start:
    mov eax,4
    mov ebx,1
    mov ecx,hello
    mov edx,helloLen
    int 80h
    mov eax,1
    mov ebx,0
    int 80h
```

Der Quellcode eines Assemblerprogrammes kann vergleichsweise einfach in binären Maschinencode übersetzt werden, da jede dort auftretende mnemonische Anweisung eineindeutig auf eine maschinenspezifische Instruktion

abgebildet werden kann.

Ausführbare Binärdatei:

Abbildung 7: Ausführbare Binärdatei (hexadezimale Ansicht)



(click on image to enlarge!)

Darüberhinaus hat der in der Graphik der [Abbildung 6](#) rechts dargestellte Ansatz der hybriden Übersetzung durch die Popularität von Programmiersprachen wie Java oder C# in jüngerer Zeit eine große Anhängerschaft erworben.

Bei dieser Vorgehensweise wird der Quellcode in ein Zwischenformat, den sog. Intermediärcode oder Bytecode, übersetzt, der dann durch einen als *Virtuelle Maschine* bezeichneten Interpreter zur Laufzeit in reale Maschineninstruktionen der Zielhardware umgesetzt wird.

Grundsätzliches Unterscheidungsmerkmal der Abarbeitung des entstehenden Maschinencodes ist die Realisierung des Instruktionssatzes. Hierbei wird zwischen CISC-, RISC- und VLIW-Varianten unterschieden, die jeweils in der Realisierung des Befehlsvorrates und der Umsetzung des Befehlsformates differieren..

CISC-Befehlssatzarchitekturen

Architekturen, die viele und hochdifferenzierte Instruktionen für vielfältige Problemstellungen anbieten werden als *Complex Instruction Set Computing* (kurz: CISC) bezeichnet.

Die Ursprünge der CISC-Architekturen reichen bis in die Zeit der Großrechner in den 1960er Jahren zurück. Damals erwies sich die Mikroprogrammierung als probates Mittel der Komplexitätsreduktion bei der Planung und Umsetzung der verarbeitenden CPUs. Überdies bargen die im ROM des Zentralprozessors abgelegten Umsetzungen mächtiger Befehle einen Geschwindigkeitsvorteil, da auf die CPU-internen Speicherbereiche in deutlich performanterer Zugriff realisiert werden konnte, als dies für die damals üblichen Ferritkern-Hintergrundspeicher erzielbar gewesen wäre.

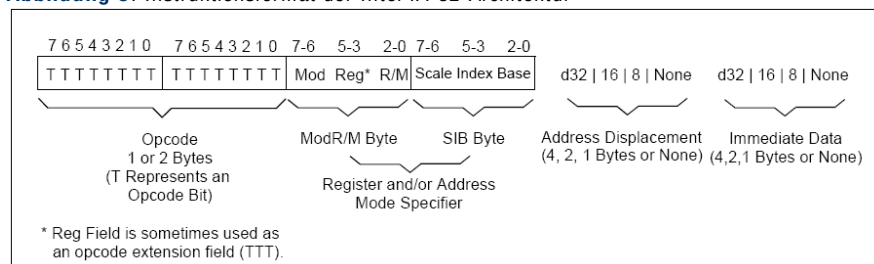
Ziel der Schaffung von Instruktionssätzen mit mehreren Hundert verschiedenen (300-400 verschiedene Instruktionstypen sind keine Seltenheit) Befehlen war es die semantische Lücke zwischen den zur Applikationsprogrammierung eingesetzten Hochsprachen und den realen Maschineninstruktionen möglichst schmal zu halten. ([[Mär01](#), S. 156f.])

Gleichzeitig verfügen CISC-Architekturen über eine Reihe verschiedener Befehlsformate, die sich in ihrem internen Aufbau unterscheiden. Im Kern bestehen Befehlsworte immer aus der Spezifikation der auszuführenden Maschineninstruktion, den benötigten Operanden und der Angabe des Speicherplatzes für das Berechnungsergebnis. Teilweise können die Eingangsoperanden direkt im Befehl untergebracht werden, statt sie aus einem Register oder einer Speicherzelle zu laden. Operanden dieses Typs werden als *Immediate Operanden* bezeichnet.

Im Falle von Verzweigungsbefehlen enthält der Operand die Adresse der nächsten auszuführenden Instruktion.

Als Konsequenz des komplexen Befehlsaufbaus und der mächtigen angebotenen Funktionalität kann die Abarbeitung eines CISC-Befehls mehrere Taktzyklen in Anspruch nehmen.

Abbildung 8: Instruktionsformat der Intel IA-32-Architektur



[Abbildung 8](#) illustriert beispielhaft das Instruktionsformat der Intel IA-32-Architektur. Es stellt den binären Opcode eines Befehls wahlweise durch ein (z.B. ADD, XOR, CMP) oder zwei Byte (z.B. MOVZX, CMPXCHG, XADD) dar, die von einigen Befehlen (z.B. MUL, DIV, NEG) durch die in den Bits 3-5 des ModR/M-Bytes untergebrachten Erweiterungsfelder zusätzlich ergänzt werden. Grundsätzlich legt das ModR/M-Byte den Adressierungsmodus eines Befehls fest, d.h. welche Register angesprochen werden (Reg-Feld) oder ob eine vorzeichenbehaftete Verarbeitung vorgenommen wird.

Als Beispiel eines Mikroprogrammes sei die Realisierung der Operation CMPXCHGBauf der Intel-Pentium-Hardware angeführt.

Die genannte Operation vergleicht den Inhalt einer 64-Bit Speicherzelle mit dem der Kombination der Register EDX und EAX. Sind die beiden Inhalte gleich, so wird der Registerinhalt aus ECX:EBX in die Speicherzelle transferiert, andernfalls der Speicherzelleninhalt in die Register geladen.

Das hierfür nötige Mikroprogramm lautet (Pseudocode, Pfeile deuten einzelne Speichertransferoperationen an):

```
IF(EDX:EAX = DEST)
    ZF <- 1
    DEST <- ECX:EBX
ELSE
    ZF <- 0
    EDX:EAX <- DEST
```

Bedingt durch die breite Verfügbarkeit schneller Hintergrundspeicher und die zunehmende Komplexität in der Erstellung der benötigten Mikroprogramme, sowie die Zunahme der zur Speicherung benötigten Chipfläche bedingte den zunehmenden Wechsel zu Befehlssätzen, deren Mächtigkeit und Umfang gegenüber dem CISC-Ansatz deutlich reduziert ausfällt.

Exkurs: Komplexitätsprobleme CISC-Architekturen: der Pentium FDIV-BUG

- [Empirische Untersuchung der Häufigkeit des Fehlerauftretens](#)
- [Statistical Analysis of Floating Point Flaw](#)

Daher integrieren vormals ausschließlich als CISC ausgelegte CPU-Familien (z.B. Intel x86, DEC-VAX) zunehmend Elemente alternativer Befehlssatzarchitekturen.

RISC-Technik

Grundidee des *Reduced Instruction Set Computings* (RISC) ist es eine Komplexitätsreduktion der CPU zur Minimierung des Umfangs des angebotenen Befehlssatzes und gleichzeitig dessen Befehlsformaten herbeizuführen.

Typische RISC-Prozessoren (wie MIPS R10000, Sun-SPARC, DEC-Alpha, Power PC) bieten daher oftmals nur ca. 50 verschiedene Befehle an, deren interne Realisierungskomplexität so stark reduziert ist, daß sie vollständig in Hardware und damit genau einem Prozessortaktyklus abgearbeitet werden können. Eine Nebenbedingung hierfür bildet die Beschränkung der in einer Maschineninstruktion zugelassenen Speicherzugriffe. So gestatten RISC-Architekturen üblicherweise lediglich Operationen auf registerresidenten Operanden, um aufwendige Speicherzugriffe zu umgehen.

Alle Elemente des Befehlssatzes eines RISC-Prozessors besitzen, unabhängig von der Wortbreite des verarbeitenden Prozessors, dieselbe Wortlänge und einen einheitlichen Aufbau.

Ausgehend hiervon vereinfacht sich der interne Prozessoraufbau und die Befehlsverarbeitung dramatisch, da keine Mikroprogramm-Unterstützung und flexible Decodierung der Befehls Worte benötigt wird. Überdies bieten RISC-Architekturen lediglich eine stark reduzierte Befehlssatzzahl an, die nur basale Instruktionen umfaßt.

Als Konsequenz dieses Reduktionsvorganges umfassen für RISC-Architekturen ausgelegte Maschinenprogramme typischerweise eine signifikant größere Anzahl Instruktionen als deren CISC-Pendant, da die RISC-Architektur die Explizierung komplexer Anweisungen auf der Maschinensprachenebene bedingt. Aufgrund des vereinfachten Prozessoraufbaus können RISC-Rechner jedoch gleichzeitig eine geringer Zykluszeit (d.h. höhere Taktfrequenz) realisieren, weshalb die in der Regel in einem Prozessorzyklus abgearbeiteten umfangreicheren Befehlsfolgen mit gegenüber der CISC-Architektur höherem Durchsatz umgesetzt werden können.

Die durch RISC-Technik erzielbare Steigerung der Verarbeitungsgeschwindigkeit wird durch die erreichbare Zykluszeit physikalisch begrenzt. Um dennoch eine weitere Skalierung zu ermöglichen werden seit den 1980er Jahren instruktionsparallele Ansätze verstärkt untersucht und auch in kommerziellen CPUs angeboten.

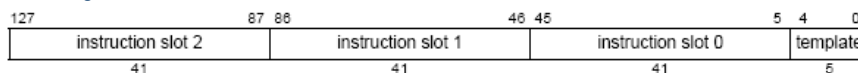
Expliziter Parallelismus und EPIC/VLIW

Architekturen mit explizitem Parallelismus auf Maschinenwortebene, sog. *Instruktionsebenenparallelität*, erlauben die simultane Verarbeitung mehr als eines Maschinenbefehls zu einem Zeitpunkt.

Voraussetzung dieser Architekturform ist die Präsenz mehrerer eigenständiger Rechenwerke, beispielsweise separater ALU-Einheiten für Integer- und Gleitkomma Berechnung.

Grundlage dieser Befehlssatzarchitekturen ist die massive Erweiterung des verarbeiteten Maschinenwortes zum *Very Large Instruction Word* (VLIW), das je nach Rechnertyp bis zu 128-Bit lange Instruktionsworte (z.B. Intels IA-64-Architektur ([Abbildung 9](#))) anbieten kann.

Begründet durch die bereits auf maschineninstruktionsebene explizierte Parallelität der Einzelinstruktionen findet sich verschiedentlich auch der durch den Hersteller Intel eingeführte Begriff des *Explicit Parallel Instruction Computings* (EPIC).

Abbildung 9: Instruktionsformat der Intel IA-64-Architektur

(click on image to enlarge!)

Der in [Abbildung 9](#) dargestellte Aufbau der IA-64-Befehlswoorte zeigt die Realisierung einer typischen VLIW-Instruktion. Jedes Befehlswort besteht aus höchstens drei eigenständigen Instruktionen, die jeweils durch 41-Bit lange (Sub-)Instruktionswoorte codiert werden. Das als *template* bezeichnete Bitfeld regelt für jedes VLIW-Instruktionswort separat die Zuordnung der Subinstruktionswoorte zu den verfügbaren Recheneinheiten bzw. spezifiziert deren Verarbeitungsnatur näher. Folgende Instruktionstypen werden unterschieden.

- ALU-Operationen
- Integer-Operationen
- Memory-Zugriffe
- Fließkommaberechnungen
- Branches (Sprunganweisungen)

Grundsätzlich sind nicht alle kombinatorisch möglichen Zusammensetzungen zugelassen. So darf beispielsweise das *slot 2* Instruktionswort spezifikationsgemäß nicht mit einem Speicherzugriff (M) bestückt werden.

Die Realisierung des Parallelismus, durch geeignete Zusammensetzung der VLIW-Worte, obliegt generell dem Übersetzer, d.h. sie wird bereits vor der tatsächlichen Ausführungszeit fixiert. Der Programmübersetzungsprozess nimmt daher -- aufgrund des zu leistenden gesteigerten Optimierungsaufwandes -- zusätzliche Zeit in Anspruch und erfordert speziell auf die Zielhardware angepasste Übersetzer.

VLIW/EPIC-basierte Ansätze können, bei geeigneten -- d.h. gut parallelisierbaren -- Problemstellungen, große Geschwindigkeitszuwächse entfalten. Ist ein Instruktionsebenenparallelismus jedoch aufgrund von starken [Datenabhängigkeiten](#) (beispielsweise wenn jede Instruktion ein durch die jeweilige Vorgängerinstruktion berechnetes (Teil-)Ergebnis als Eingabe erwartet) nicht möglich, so werden müssen viele Subinstruktionswoorte durch Leeroperationen (NOP) aufgefüllt werden und die durch sie ansprechbaren Funktionseinheiten bleiben ungenutzt.

Grundsätzlich lassen sich Befehlsfolgen ohne [Abhängigkeiten](#), wie nebenläufig auszuführende Threads sehr effizient in VLIW-Instruktionswoorte.

Web-Referenzen 1: Weiterführende Links

- [Univac History](#)
- [The History of the UNIVAC Computer](#)
- [UNIVAC Memories](#)
- [Hello-World-Programm auf IBM 1401](#)
- [MULTICS](#)
- [Computer Model Katalog](#)
- [Rechner- und Betriebssystemgenerationen und deren Auswirkungen auf betriebliche Anwendungssysteme](#)
- [Moore's Gesetz](#)
- [Two Approaches to 64-Bit Computing](#)



2.2 Mikroarchitekturen

Parallel zum Übergang von CISC- zu RISC-basierten Befehlsarchitekturen und der zunehmenden Verbreitung von auf dem VLIW-Ansatz aufbauenden Prozessoren werden neue Leistungspotentiale auch immer mehr durch neue Konzepte auf der Mikroebene erschlossen. Diese Ebene ist hierbei unterhalb der Abarbeitung der einzelnen Instruktionen der Befehlsarchitektur angesiedelt und widmet sich ausschließlich der internen Realisierung der Verarbeitung einer einzelnen Maschineninstruktion.

Phasen-Pipelining

Insbesondere bei RISC-Architekturen, die sich zum Ziel setzen jede Maschineninstruktion in Schnitt in genau einem Taktzyklus abarbeiten, definieren die physikalischen Signallaufzeiten eine Obergrenze des Prozessortaktes, da in einem Takt nicht beliebig viele logische Operationen ausgeführt werden können.

Um dennoch die Verarbeitungsgeschwindigkeit einerseits und gleichzeitig die Auslastung der verschiedenen Prozessorkomponenten andererseits zu erhöhen etablierten sich *Phasen-Pipeline-Architekturen*. Grundansatz dieses Mikroarchitekturtyps ist es, jeden Einzelbefehl in eine Reihe fein granulierter Mikroinstruktionen aufzuspalten und diese durch jeweils durch separate CPU-Komponenten abzuarbeiten. Als Resultat können gleichzeitig Befehle verschiedener Ausführungsphasen verarbeitet werden. Die hierbei abzuarbeitenden Einzelverarbeitungsschritte sind in jedem Falle von deutlich geringerer Schaltungskomplexität als vollständige Befehlsverarbeitungen.

Abbildung 10: Schema einer sechsstufigen Phasen-Pipeline

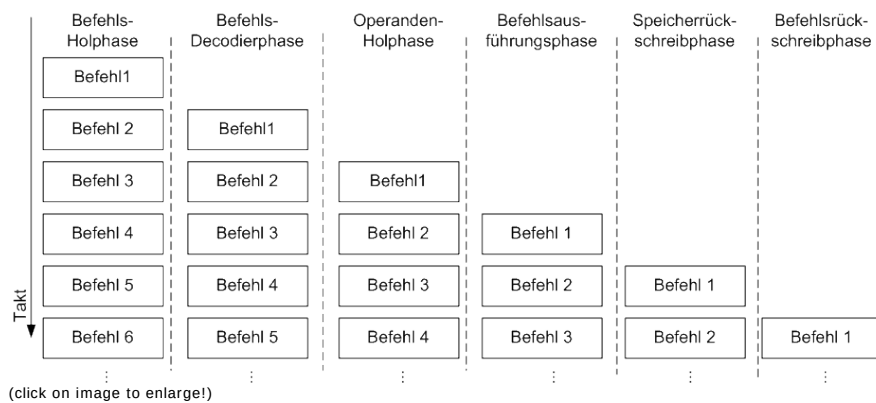


Abbildung 10 illustriert den Aufbau einer sechsstufigen Phasen-Pipeline. Ihre Nutzung setzt voraus, daß die Ausführung jedes Maschinenbefehls in die sechs dargestellten Phasen eingeteilt wird:

- **Befehlsholphase (IF)**
Der nächste auszuführende Maschinenbefehl wird durch das Leitwerk in die ALU geladen. Anschließend wird der Befehlszähler erhöht.
- **Befehlsdecodierphase (DE)**
Die für den geladenen Maschinenbefehl auszuführenden Mikroprogramm-Operationen werden ermittelt.
- **Operandenholphase (OF)**
Die zur Befehlsausführung benötigten Eingangsdaten (Operanden) werden geladen.
- **Befehlsausführungsphase (EX)**
Durch Befehlsausführung wird das Ergebnis berechnet.
- **Speicherrückschreibphase (MEM)**
Berechnungsergebnisse werden in Speicher zurückgeschrieben.
- **Befehlsrückschreibphase (WB)**
Das Berechnungsergebnisse werden in Register zurückgeschrieben.

Wird der Füllungsgrad der Phasen-Pipeline betrachtet, so fällt auf, daß jede Pipeline erst nach einer gewissen Anzahl Takten Ergebnisse liefert. Wird für jede Phase eine Ausführungszeit von genau einem Taktzyklus unterstellt, so ist die *Einschwingphase* identisch zur Pipelinelänge. Daher liefert die Pipeline der [Abbildung 10](#) ihr erstes Ergebnis erst nach sechs Takten.

Alle darauffolgenden Takte wird jedoch im Idealfalle ein weiteres Ergebnis produziert.

Der maximale Auslastungsfall, der in jedem Taktzyklus ein Ergebnis berechnet, kann jedoch nur erreicht werden, wenn ein gleichmäßiger Füllungsgrad der Pipeline vorliegt, d.h. alle Pipelineinstufen befüllt sind. Dies setzt jedoch voraus, daß keine Datenabhängigkeiten zwischen den verarbeiteten Einzelbefehlen bestehen, da zu Auslastungslücken führen können.

Solche Auslastungslücken (sog. *pipeline bubbles*) treten dann auf, wenn bestimmte Abhängigkeitssituationen (sog. *pipeline hazards*) gegeben sind, die eine effiziente Nutzung der Pipeline verhindern.

Typischerweise werden drei Typen von Pipeline Hazards unterschieden:

- **Ressourcen Abhängigkeit** treten dann auf, wenn nicht alle Phasen beliebig überlappend ausgeführt werden können.
- **Datenabhängigkeit** treten dann auf, wenn eine Instruktion das Ergebnis der direkt vorhergehenden als Eingangsoperand benötigt.
- **Kontrollflußabhängigkeit** treten dann auf, wenn die Verarbeitung einer Instruktion durch Modifikation des [Befehlszeigers](#) einen anderen als den bereits in der Pipeline befindlichen als nächste auszuführende Instruktion wählt.

Beispiel 1: Ressourcen Abhängigkeit

Instruktion		Taktzyklus									
		1	2	3	4	5	6	7	8	9	
	I ₁	IF	DE	OF	EX	MEM	WB				
	I ₂		IF	DE	OF	EX	MEM	WB			
	I ₃			IF	DE	OF	EX	MEM	WB		
	I ₄				IF	DE	OF	EX	MEM	WB	
	I ₅					IF	DE	OF	EX	MEM	

Beispiel 1 zeigt als Beispiel einer Ressourcenabhängigkeit den Zustand der sechsstufigen Beispielpipeline, der im Falle nur genau eines Busses (oder nur genau einer Kontrolleinheit) zum Speicherzugriff eintritt. Hierbei kann die *Befehlsholphase* nicht überlappend mit der *Speicherrückschreibphase* ausgeführt werden.

Zur Lösung dieses Problems werden Warteanweisungen (NOP) in die Pipeline eingesteuert, während der die Verarbeitung in der betreffenden Stufe ruht.

Das Ergebnis ist in [Abbildung 2](#) dargestellt. Auffallend ist hierbei, daß selbst nach Einfügen einer NOP-Operation der

Befehlsholvorgang für die Instruktion I_5 nicht durchgeführt werden kann, da sich dieser mit dem Speicherzugriff des 2. in der Verarbeitung befindlichen Befehls überschneiden würde. Daher wird erneut ein Wartezyklus eingefügt und die Verarbeitung der 5. Instruktion verzögert.

Dieser Vorgang wiederholt sich bis im 9. Verarbeitungstakt keine Überlappung zwischen Befehlsholphase und Speicherrückschreibphase mehr auftritt.

Beispiel 2: Auflösung der Ressourcen-Abhängigkeit

Instruktion	Taktzyklus								
	1	2	3	4	5	6	7	8	9
I_1	IF	DE	OF	EX	MEM	WB			
I_2		IF	DE	OF	EX	MEM	WB		
I_3			IF	DE	OF	EX	MEM	WB	
I_4				IF	DE	OF	EX	MEM	WB
(I_5-I)					NOP	IF	DE	OF	EX
(I_5-II)					NOP	NOP	IF	DE	OF
(I_5-III)					NOP	NOP	NOP	IF	DE
I_5					NOP	NOP	NOP	NOP	IF

Beispiel 3 zeigt den Fall einer Datenabhängigkeit, die durch den Verarbeitungsversuch des Ergebnisses der Addition durch die nachfolgende Subtraktion entsteht.

Beispiel 3: Datenabhängigkeit

Instruktion	Taktzyklus						
	1	2	3	4	5	6	7
ADD R1,R2,R3	IF	DE	OF	EX	MEM	WB	
SUB R4,R5,R1		IF	DE	OF			

Durch Einstreuung von NOP-Befehlen kann die Verarbeitung der Subtraktion solange verzögert werden, bis das Berechnungsergebnis der vorangehenden Addition zur Verfügung steht. So kann die Verarbeitung erst nach drei Wartezyklen mit der Operandenholphase fortgesetzt werden.

Beispiel 4: Lösung der Datenabhängigkeit

Instruktion	Taktzyklus						
	1	2	3	4	5	6	7
ADD R1,R2,R3	IF	DE	OF	EX	MEM	WB	
(SUB R4,R5,R1)		IF	DE	NOP			
(SUB R4,R5,R1)		IF	DE	NOP	NOP		
(SUB R4,R5,R1)		IF	DE	NOP	NOP	NOP	
SUB R4,R5,R1		IF	DE	NOP	NOP	NOP	OF

Das Schema des Beispiels 5 zeigt eine Datenflußabhängigkeit, die durch den bedingten Sprung JNE (Verzweigung bei Ungleichheit) entsteht. Nach Ausführung des Sprungbefehls muß bei I_1 fortgesetzt werden. Dies geschieht durch Umsetzen des Befehlszeigers unter Überspringen der auf die JNE-Instruktion folgenden Anweisungen. Diese sind jedoch bereits in die Phasen-Pipeline geladen und liefern daher die nicht mehr benötigten (rot dargestellten) Ergebnisse.

Beispiel 5: Kontrollflußabhängigkeit



Instruktion	Taktzyklus								
	1	2	3	4	5	6	7	8	9
CMP R1,0	IF	DE	OF	EX	MEM	WB			
JNE I1		IF	DE	OF	EX	MEM	WB		
...			IF	DE	OF	EX	MEM	WB	
...				IF	DE	OF	EX	MEM	WB
...					IF	DE	OF	EX	MEM
I1: ...							IF	DE	

Die Einsteuerung von NOP-Anweisungen, bzw. das Verwerfen nicht mehr benötigter Ergebnisse, garantiert zwar die Korrektheit der berechneten Resultate, setzt jedoch die Leistungsfähigkeit einer Phasen-Pipeline herab. So erzwingt die restriktive Umsetzung des Speicherzugriffes aus Beispiel1 zum Einschub von vier Wartezyklen nach jeweils vier Berechnungsphasen.

Ähnliches gilt für die durch Beispiel3 betrachteten Datenabhängigkeiten. Auch sie können nur durch Einbringung entsprechender Verzögerungen, während der die ALU nicht genutzt wird, aufgelöst werden.

Im Falle der Kontrollfußabhängigkeit werden zwar keine Wartezyklen benötigt, jedoch werden im Verlauf der Verarbeitung durch die Pipeline Ergebnisse berechnet, die nicht benötigt werden, was effektiv einer Zeitverzögerung in der Bereitstellung der tatsächlich gewünschten Resultate gleichkommt.

Phasenpipelines in der Praxis

Die Verwendung von Phasenpipelines zur Steigerung des Durchsatzes bei gleichzeitiger Erhöhung der Taktfrequenz hat sich inzwischen breit bei kommerziellen Mikroprozessoren durchgesetzt.

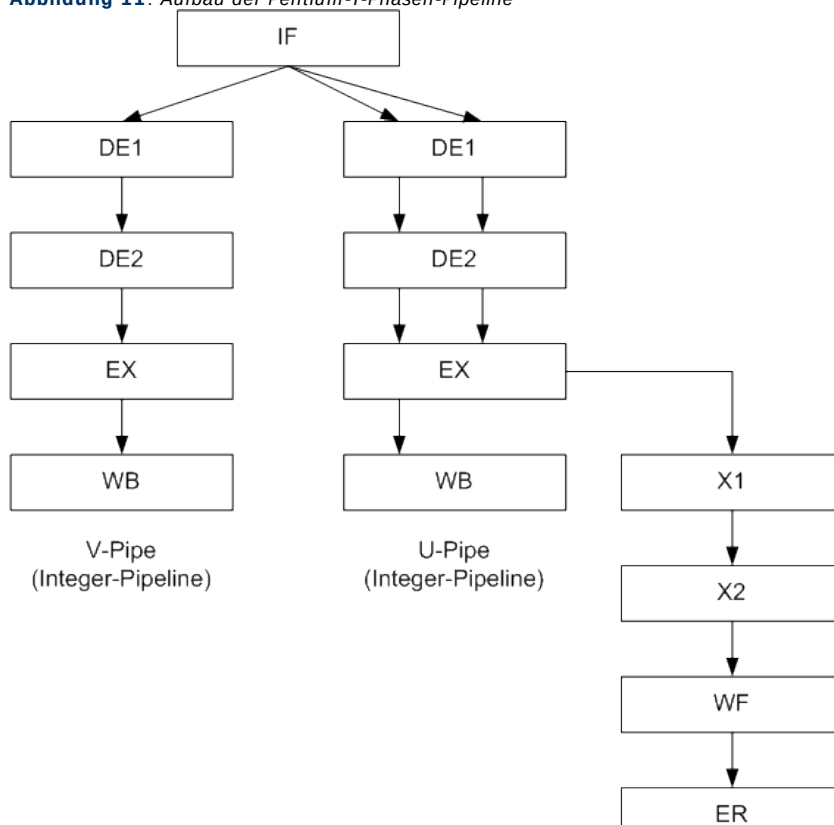
Intel verwendet seit der Pentium-Architektur Pipelining-Techniken, die zuerst eher zaghaft durch die maximal achtstufige Umsetzung in den Pentium-I-CPUs Einzug hielt, jedoch inzwischen mit einer extrem großen 30-stufigen Umsetzung in den Pentium-III-Chips bzw. 20 Stufen beim Pentium-4 ihre Entfaltung findet. Diese AMD verwendet im 32-Bit Athlon-Prozessor eine 10-stufige Pipeline und bietet in der 64-Bit CPU-Variante eine um zwei zusätzliche Stufen erweiterte Umsetzung an.

Die ungewöhnlich lange Pipeline der Pentium-III- und -4-Realisierungen zeigt, daß auch große Phasenpipelines effizient eingesetzt werden können und erlauben es die genannten CPUs bei sehr hohen Taktraten zu betreiben.

Die [Abbildung 1](#) zeigt die Realisierung der ersten Generation der Pentium-Mikroarchitektur, welche über eine zweifach gegabelte Pipeline verfügt. Die Architektur verfügt über drei parallel angeordnete Pipelines, wovon die als „U-“ und „V-“ Pipeline bezeichneten jeweils zur Verarbeitung von ganzzahligen Ausdrücken dienen. Zusätzlich zweigt die Ausführungsphase (EX) der U-Pipeline in die FP-Pipeline ab, welche vier zusätzliche Stufen zur Verarbeitung von Fließkommawerten bereitstellt.

Neben den [bereits bekannten](#) Pipelinephasen zeigt die Graphik die zusätzlichen Ausführungsphasen X1 und X2, sowie die gesonderte Rückschreib- (Write Float (WF)) und Fehlerbehandlungsphase (Error Handling (ER)) für Fließkommawerte.

Abbildung 11: Aufbau der Pentium-I-Phasen-Pipeline



FP-Pipe (Fließkomma-Pipeline)

(click on image to enlarge!)

Durch die extrem langen Pipelines gängiger Prozessoren werden sehr hohe Anforderungen an die adäquate Bereitstellung der zu verarbeitenden Instruktionen gestellt, da sich Pipelinekonflikte mit gravierenden Performanceeinbußen auswirken können.

Aus diesem Grunde führen *superskalare* Architekturen eine Reihe von Maßnahmen zur Verbesserung der Pipelineauslastung ein.

Web-Referenzen 2: Weiterführende Links



- [Pipeline Hazards](#)
- [Pipelining 101](#)
- [Pentium-4-Hyper-Pipeline](#)
- [The perils of deep pipelining](#)
- [The Anatomy of Modern Processors: Pipelines](#)
- [Pipeline Streaming](#)
- [Pipelines of Superscalar \(Pentium Family\) and Dynamic Execution \(P6-Family\) Architectures](#)

Superskalarität

Superskalare CPUs versuchen den Durchsatz durch CPU-interne Parallelität zu erhöhen. Hierzu werden Funktionseinheiten derselben Funktionseinheiten physisch mehrfach realisiert und angeboten. Zur Versorgung der Einzeleinheiten können entweder explizite [VLIW-basierte Ansätze](#) bereits zur Übersetzungszeit oder dynamische Pipeline-basierte zur Ausführungszeit Einsatz finden.

Nachfolgend wird ein Überblick der Strategien zur Ansteuerung von Funktionseinheiten in superskalaren CPUs gegeben. Gleichzeitig werden Strategien und Techniken diskutiert, die eingesetzt werden können um die Anzahl der Wartezyklen innerhalb einer Pipeline zu verringern und so ihre Auslastung zu steigern.

• Hardwaremaßnahmen

Zur Behebung struktureller Ressourcen-Konflikte können zusätzliche Hardwarebausteine wie parallele Speicherzugriffskanäle und -controller eingesetzt werden um die zwangsweise Serialisierung der Instruktionen an diesen „Engstellen“ (engl. *bottle neck*) zu verhindern.

• Kontrollfluß-Modifikationen

Zur Erhöhung der Auslastung einer gegebenen CPU kann die Verarbeitungsreihenfolge der Instruktionen eines Programms von der Reihenfolge ihres Auftretens im Programm abweichen. Hierbei werden zwei Verarbeitungstypen unterschieden:

◦ In-order-Execution

Hierbei wird die Reihenfolge der im Programm codierten Befehle zunächst nicht verändert, sondern durch gegebene Pipelines auf verschiedene Ausführungseinheiten verteilt und so parallelisiert. Bekanntestes Beispiel für diesen Ansatz ist die [Pipeline des Pentium](#).

◦ Out-of-Order-Execution

Instruktionen können gleichsam „auf Vorrat“ ausgeführt werden um den Leerlauf, der durch NOP-Operationen entsteht, produktiv zu nutzen.

• Registersatz-Modifikationen

Die während der Out-of-order-Ausführung entstehenden (noch) nicht benötigten Berechnungsergebnisse werden -- um störende Wechselwirkungen mit den regulär verarbeiteten Instruktionen auszuschließen -- in separaten Registern (sog. *Schattenregistern*) abgelegt.

• Sprungvorhersage

Um die Produktion nicht mehr benötigter Resultate möglichst auszuschließen wird durch zusätzlichen Aufwand eine Sprungvorhersage betrieben, die es ermöglicht (mit einer gewissen Wahrscheinlichkeit) das Sprungziel vorherzubestimmen und die (vermutlich) benötigten Befehle „auf Vorrat“ in die Pipeline zu laden.

• Compilermaßnahmen

Ist der verwendete Übersetzer auf die später verwendete Zielarchitektur abgestimmt, so können gewisse vorausschauende Optimierungen bereits vor der Laufzeit vorgenommen werden.

Neben dem Einsatz superskalarer Techniken zur Steuerung des Durchsatzes der CPU hat sich in jüngerer Zeit *simultanes Multithreading* zur weiteren Leistungssteigerung am Markt etabliert.

Multithreading-CPU's

Multithreading-CPU's führen die Unterstützung nebenläufig ausführbarer Programmteile noch einen Schritt weiter über die Möglichkeiten des [expliziten Parallelismus auf Instruktionsebene](#) hinaus und nehmen Anleihen bei den symmetrischen Multiprozessorsystemen, welche in einem System mehrere eigenständige CPU's vereinen.

Um jedoch der Kostensituation und verschiedenen Einsatzproblemen von Multiprozessorsystemen zu begegnen bieten Multithreading-CPU's nur simultane Instruktionsverarbeitung an ohne alle Ausführungsressourcen (wie Registersätze) mehrfach zu realisieren.

Grundsätzlich steuert in einer Multithreading-CPU genau ein Leitwerk mehrere getrennt ansprechbare Recheneinheiten an. Auf diese Weise können sich in einem Taktzyklus mehrere Instruktionen in der [Befehlsausführungsphase](#) befinden und Ergebnisse liefern.

Bei Multithreading-CPU's vergrößert sich tendenziell der Grad der Konkurrenz um die verfügbaren Ressourcen wie Register, Zwischenspeicher (Caches) und andere prozessorinterne Verwaltungseinheiten, so daß erhöhter Aufwand für

die Synchronisation und Behebung der auftretenden Zugriffskonflikte anzusetzen ist.

Grundsätzlich werden (nach [Mär01, S. 250] mit Verweis auf Culler) drei Varianten der Multithreading-Realisierung in einer CPU unterschieden:

- **Blocked Multithreading:**

Eine begrenzte Anzahl zusätzlicher Register wird bereitgestellt, jedoch keine zusätzlichen Verarbeitungsressourcen. Als Konsequenz muß ein rechenbereiter Thread warten, bis ihm eine freie Rechenkomponente zugeteilt wird.

- **Interleaved Multithreading:**

In jedem Taktzyklus wird erneut ein Thread zur Ausführung ausgewählt. Ein gegebener Befehl muß auf seine Ausführung daher so lange warten, bis er zur Abarbeitung selektiert wird.

- **Simultaneous Multithreading(SMT):**

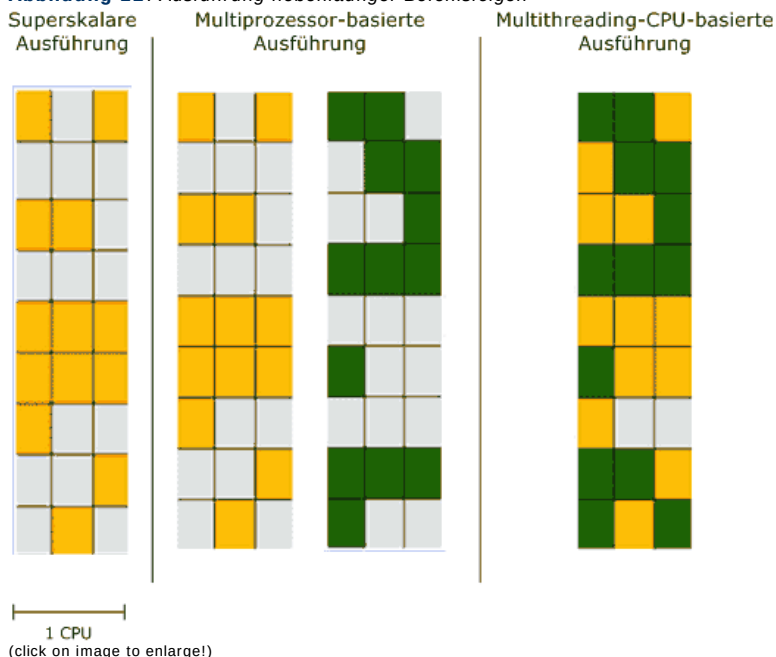
Kombiniert interleaved Multithreading und superskalare Ansätze um die gleichzeitige Ausführung von Instruktionen in einem Taktzyklus ermöglichen zu können.

[Abbildung 12](#) zeigt die Ausführung nebenläufiger Befehlsfolgen auf einer superskalaren, einer Multiprozessor- und einer Multithreading-CPU.

Während die superskalare CPU die vorhandenen Ressourcen durch die auftretenden Wartezyklen nicht vollständig ausnutzen (48% ungenutzter Anteil im Beispiel) kann das Multiprozessorsystem zwar den Durchsatz des Gesamtsystems durch die parallele Ausführung separater Befehlsfolgen erhöhen bietet jedoch prozentual nur in etwa dieselbe Auslastung der vorhandenen Ressourcen.

Im Idealfalle kann eine Multithreading-basierte CPU, durch geeignete Zuteilung der Berechnungsaufgaben an die vorhandenen Berechnungsressourcen einen höheren Gesamtdurchsatz der CPU erreichen.

Abbildung 12: Ausführung nebenläufiger Befehlsfolgen



Gegenwärtig wird SMT durch Intel in seiner Pentium-4-Architektur popularisiert und für den Massenmarkt angeboten. Erste Erfahrungen mit diesem CPU-Typ stimmen durchaus positiv, wenngleich der Geschwindigkeitszugewinn sich lediglich im Bereich von 30% bewegt, was unter anderem auf die fortbestehende Konkurrenz um verschiedene Prozessor-Ressourcen zurückzuführen ist.

Web-Referenzen 3: Weiterführende Links



- [Hyperthreading](#)
- [Hyper-Threading Technology](#)
- [Hyperthreading @ Intel](#)
- [SMT vs. SMP](#)

2.3 Parallelverarbeitung

Rechnerklassifikation

Aspekte der Parallelverarbeitung werden in aktuellen Rechnerarchitekturen auf verschiedenste Weise und unterschiedlichen Ebenen genutzt, so daß eine trennscharfe Unterscheidung für gegenwärtige Prozessoren kaum mehr möglich ist, da auch diese bereits Parallelitätstechniken einsetzen.

Klassischerweise werden nach Flynn die vier in [Abbildung 13](#) dargestellten Verarbeitungsformen unterschieden, die jedoch heute nicht mehr in ihrer ursprünglichen Reinform auftreten:

- **SISD-Rechner:**

In diese Klasse fallen die mit genau einem Leiterwerk und einem Rechenwerk ausgestatteten klassischen Universalrechner.

SISD-Rechner führen zu einem gegebenen Zeitpunkt daher genau einen Befehl auf einem verarbeiteten Datum aus.

Die gegebene Klassifikation läßt jedoch überlappende Ausführung, wie sie beispielsweise durch [Phasen-Pipeline](#) entstehen ebenso außer Acht, wie Parallelität auf Ebene der Rechenwerke wie sie durch [Multithreading-CPU](#)

eingeführt wird.

- **SIMD-Rechner:**

In diese Klasse fallen alle mit genau einem Leitwerk und mehreren Rechenwerken ausgestatteten Feldrechner oder Array-Prozessoren.

Durch die Duplizität der Rechenwerke gestatten SIMD-Rechner die parallele Ausführung jeder Einzelinstruktion auf verschiedenen Eingangsdaten zu einem Zeitpunkt. Jedoch muß auf allen verarbeiteten Daten jeweils dieselbe Instruktion zur Ausführung gebracht werden.

Typische Anwendungsfelder dieses Rechnertyps umfassen die Verarbeitung multimedialer Bild- und Tondaten.

Auch Vektorprozessoren, die Operanden, welche aus einer festen Anzahl von Skalarwerten bestehen, verarbeiten können gemäß Gilois der Klasse der SIMD-Rechner zugeordnet werden.

- **MISD-Rechner:**

Dieser Rechnertypus -- der mehrere Leitwerke, aber nur ein Rechenwerk vorsehen würde -- ist in der Praxis bisher nicht anzutreffen. Eine mögliche Realisierung dieser Rechnerklasse müßte ein „fließbandartig“ betriebenes Rechenwerk mit Daten, die durch unterschiedliche Leitwerke bereitgestellt werden, beschicken.

- **MIMD-Rechner:**

In diese Klasse fallen alle Rechner, welche über mehrere Leit- und mehrere unabhängige Rechenwerke verfügen. Diesem Rechnertypus sind alle Formen von Parallelrechnern mit mehreren eigenständigen CPUs sowie verschaltete Rechner zuzuordnen, sofern sich diese als genau ein virtuelles System präsentieren.

Abbildung 13: Rechnerklassifikation nach Flynn

		Data	
		Single	Multiple
Instruction	Multiple	MISD	MIMD
	Single	SISD	SIMD

(click on image to enlarge!)

Das Flynn'sche Schema stößt bereits bei der Rubrizierung aktuell verfügbarer Prozessoren, wie den mit mehreren separaten Funktionseinheiten ausgestatteten Intel- oder AMD-CPU's an seine Grenzen. Zwar kann jede einzelne Funktionseinheit der Klasse SIMD zugeordnet werden, jedoch ist eine eindeutige Klassifikation der Gesamt-CPU nicht mehr unstrittig möglich.

Speichergekoppelte-Umgebungen

In speichergekoppelten Umgebungen steht allen physischen Prozessoren ein gemeinsamer Hauptspeicher zur Verfügung. Anhand der physischen Realisierung dieses Speichers wird zwischen Architekturen des Typs *Unified Memory Access (UMA)* und *Non-Unified Memory Access (NUMA)* unterschieden.

Die enge Kopplung zwischen den Einzelprozessoren der UMA-Realisierungen setzt voraus, daß alle Prozessoren durch leistungsfähige Busse Zugang zum gemeinsam genutzten Speicher erhalten. Daher ist der physische Abstand zwischen jedem Einzelprozessor und dem Hauptspeicher in der Regel identisch groß.

Aufgrund der sich daraus inhärent ergebenden Sternarchitektur wird dieser Parallelrechnertyp auch als *Symmetrisches Multiprocessing* bezeichnet.

Dieser Rechnertyp ist in der Praxis -- mit Ausbaustufen von bis zu acht verschalteten Prozessoren -- häufig anzutreffen. Die Integration größerer CPU-Anzahlen sind in Ihrer Realisierung als problematisch anzusehen, da die Konkurrenz um den Hauptspeicher bei steigender CPU-Anzahl zu überproportionalen Leistungseinbußen führt. Zusätzlich kann der Zugewinn an Busbandbreite nicht mit der Leistungssteigerung der CPUs Schritt halten, weshalb sich jeder Speicherzugriff als Performance aufzehrende Engstelle erweist.

Durch lose Kopplung in NUMA-Umgebungen, welche inhärent die unterschiedliche Realisierung der Speicherzugriffe für alle angebundenen CPUs zulassen können diese Engpässe vermieden werden. Hierzu kann an jedem physischen Knoten eigener Hauptspeicher angesiedelt werden. Die einzelnen „Hauptspeicherinseln“ werden durch zusätzliche Hardware oder das das verwaltende Betriebssystem zu einem virtuellen Gesamtspeicher verschaltet.

Nachrichtengekoppelte-Umgebungen

Wird die Forderung nach gemeinsamem Speicher zwischen den Einzelknoten aufgegeben, so lassen sich flexiblere Umsetzungen finden, die auch deutlich höhere CPU-Anzahlen umfassen können.

Jedoch entfällt dann der gemeinsame Speicher als Medium des Datenaustausches zwischen den Einzelprozessoren und muß durch eine explizite Kommunikationsinfrastruktur ersetzt werden.

Hierbei kommen typischerweise Ansätze der Botschaftenvermittlung (*message passing*) zum Einsatz, welche Datenpakete zwischen den beteiligten Prozessorknoten versenden.

Eine bekannte Umsetzung in diesem Umfeld stellt das [Message Passing Interface](#) dar, welches eine Programmiersprachen-Bibliothek zur Verfügung stellt, welche die schnelle Botschaftenvermittlung zwischen verteilt ausgeführten Fortran- oder C-basierten Applikationen ermöglicht.

Leistungsabschätzung

Die zu erwartende Leistungssteigerung (*speedup*) eines Multiprozessorsystems kann durch verschiedene Gesetze abgeschätzt werden. Bekanntester Ansatz hierzu ist das [Amdahlsche Gesetz](#).

In seiner in [Abbildung 14](#) abgebildeten Form setzt die parallel (p_{parallel}) und seriell (p_{seriell}) auszuführenden Programmteile, sowie die Anzahl der verfügbaren CPUs (n_{CPU}) für eine feste Problemgröße in Beziehung und ermittelt daraus statisch die zu erwartende Leistungssteigerung (S).

Für den Zusammenhang zwischen seriell und parallel ausführbarem Code gilt dabei: $p_{\text{seriell}} + p_{\text{parallel}} = 1$.

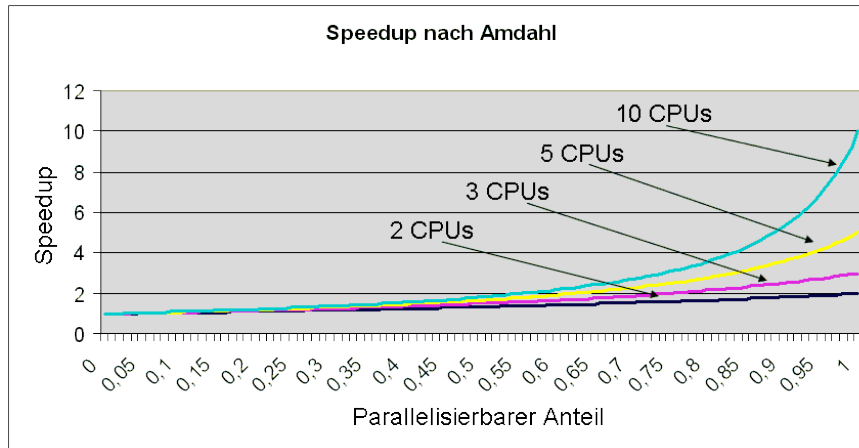
Abbildung 14: Amdahlsches Gesetz

$$S := \frac{1}{p_{\text{seriell}} + \frac{p_{\text{parallel}}}{n_{\text{CPU}}}}$$

(click on image to enlarge!)

[Abbildung 15](#) zeigt den abgeschätzten Geschwindigkeitsgewinn relativ zu den betrachteten Freiheitsgraden.

Abbildung 15: Abschätzung des Speedups nach Amdahl



(click on image to enlarge!)

2.4 Fallstudie: Die virtuelle Rechnerarchitektur der Java-Maschine

Kern der oft apostrophierten [Plattformunabhängigkeit](#) der Programmiersprache Java ist die Generierung eines generischen Zwischenformates -- des Byte-Codes. Dieser wird von einer plattformabhängig implementierten Programmeinheit, der *Java Virtual Machine* (Abk. JVM) interpretativ zur Ausführung gebracht.

Jede Java-Applikation wird auf einer eigenen virtuellen Maschine zur Ausführung gebracht. Dies garantiert eine größtmögliche Abschottung, mit dem Ziel maximierter Sicherheit, der möglicherweise gleichzeitig auf einer realen Maschine ausgeführten Java-Programme voneinander.

Das Konzept der virtuellen Maschine, die als Programm auf einer realen Hardware abläuft, ermöglicht eine vergleichsweise einfache und schnelle Portierbarkeit auf neue Zielumgebungen, da lediglich die virtuelle Maschine an die veränderte reale Maschine angepaßt werden muß.

Der Gedanke virtueller Maschinen, die generischen Zwischencode -- oftmals auch als *P-Code* bezeichnet -- ausführen, ist nicht neu. Bereits USCD Pascal, E-BASIC und die verschiedenen SmallTalk-Implementierungen, setzt diesen praktisch um.

Die Realisierung der Befehlsfolgen (Opcodes) innerhalb der virtuellen Maschine von Java ähnelt teilweise frappant der Architektur der für die Züricher Pascal-Implementierung entwickelten (abstrakten) P-Maschine.

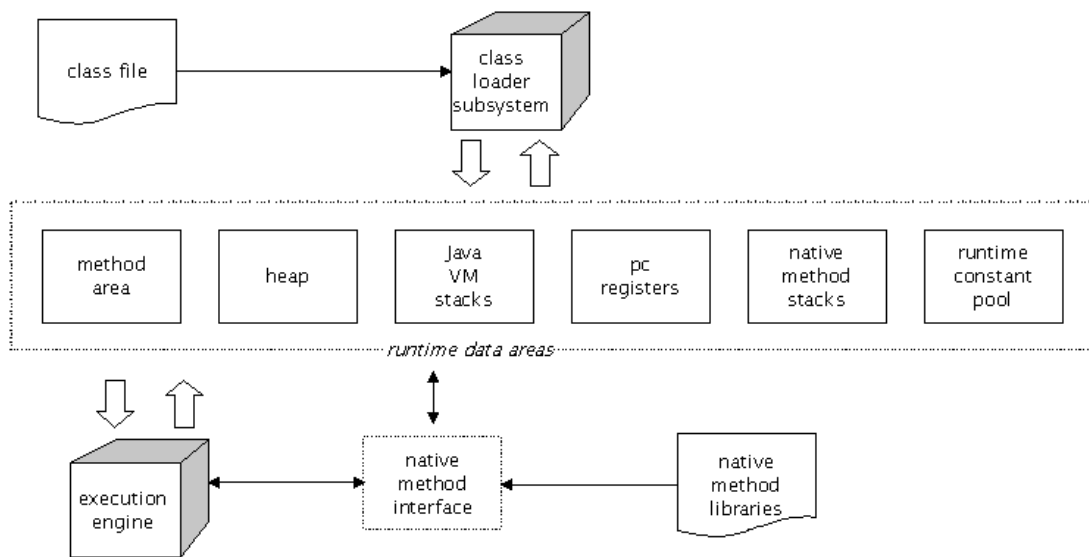
Inzwischen steht mit der [zAAP](#) (zSeries Application Assist Processor)-Hardware für die IBM-Mainframemaschinen z890 und z990 sogar eine vollständige Hardwareimplementierung der JVM zur Verfügung, welche den Charakter der *virtuellen* zu einer *realen* Maschine weiterentwickelt.

Ein Beispiel einer vollständig als Software realisierten „klassischen“ virtuellen Maschine ist [java](#), die Bestandteil des Java-Development Toolkits von SUN ist.

Bekannte andere virtuelle Maschinen sind: [Kaffe](#) oder auch [IBMs Jikes-Implementierung](#)

[Wie bereits bekannt](#) wird eine Java-Applikation durch den Aufruf `java`, gefolgt vom Namen der Startklasse und etwaiger Kommandozeilenparameter ausgeführt. Technisch gesehen bewirkt der Aufruf zunächst die Erzeugung einer neuen Instanz der virtuellen Maschine, auf welcher die Programm-Abarbeitung mit der [main-Methode](#) der Startklasse begonnen wird.

Eine Instanz einer virtuellen Maschine existiert, solange Programmfäden (engl. *Threads*) (genaugenommen: non-daemon Threads) ausgeführt werden, bzw. die virtuelle Maschine explizit beendet wird (mit dem API-Aufruf [System.exit\(\)](#)) oder ein Fehler auftritt.



Die wesentlichen Bestandteile der JVM sind:

- **method area**
Einmal vorhanden je virtueller Maschine, wird gemeinsam von allen Threads benutzt; enthält klassenspezifische Daten (Typinformation).
- **heap**
Einmal vorhanden je virtueller Maschine, wird gemeinsam von allen Threads benutzt; enthält die dynamisch erzeugten Objekte.
- **Java VM stacks**
Threadspezifisch. Enthält Zustand von nicht-nativen Methodenaufrufen, deren lokale Variablen, Übergabeparameter, den möglichen Rückgabewert sowie Methoden-interne Berechnungsergebnisse. ([vgl. JVM-Spezifikation](#))
- **pc registers**
Threadspezifisch; enthält für jeden Zeitpunkt der Ausführung einer nicht-nativen Methode die Adresse der derzeit ausgeführten Instruktion. ([vgl. JVM-Spezifikation](#)). Während der Abarbeitung nativer Methoden ist der Wert auf undefined gesetzt. Konkrete Größe des virtuellen pc-Registers hängt von der Adresslänge der realen Plattform ab.
- **native method stack**
Implementierungsspezifischer Speicherbereich zur Behandlung von nativen Methodenaufrufen.
- **runtime constant pool**
Klassen- oder Schnittstellenspezifisch. Enthält verschiedene Konstanten, die zur Übersetzungszeit feststehen und in der constant_pool Tabelle der class-Datei abgelegt sind. Die Funktion dieses Bereichs ähnelt dem einer konventionellen Symboltabelle. ([vgl. JVM-Spezifikation](#))

Die Java-Stacks sind in *stack frames* organisiert. Jedem Methodenaufruf ist ein [Stack-Frame](#) zugeordnet, der beim Aufruf erzeugt (push), und beim Verlassen (pop) vom Stack entnommen wird. Innerhalb eines Frames befindet sich

- Array der lokalen Variablen
- Operanden-Stack
Wichtigste Datenstruktur innerhalb der virtuellen Maschine. Der Operanden-Stack enthält alle Eingangs- und Ausgangsdaten der verschiedenen Berechnungen. Seine Einträge sind typisiert in 32-Bit Worten organisiert; die korrekte Typisierung der Eingangsoperanden wird von jedem Opcode geprüft.
- Referenz auf runtime constant pool
- implementierungsspezifische- und debugging-Information

Den für den Anwendungsentwickler offensichtlichsten Bestandteil der virtuellen Maschine, bilden jedoch die JVM-Instruktionen -- die Maschinensprache der JVM.

Der [Befehlssatz der JVM](#) umfaßt ausschließlich genau ein Byte lange Opcodes.

Die JVM ist generell *stack-orientiert*. Dies bedeutet, daß Quell- und Zielooperanden der meisten Operationen werden vom Stack entnommen, und das Ergebnis dort abgelegt. Insbesondere existieren, abgesehen von vier Verwaltungsspeicherplätzen je Ausführungs-Thread, keine virtuellen Prozessorregister, um die Implementierungsanforderungen an die reale Plattform zu minimieren.

Als threadlokale Register stehen zur Verfügung:

- [pc -- program counter](#)
Enthält die Adresse der gerade ausgeführten Instruktion. Handelt es sich um eine native-Methode, die in einer anderen Programmiersprache ausgeführt wird, so ist der Wert *undefined*.
- [optop](#)
Verweist auf die Spitze des [Operanden Stacks](#).
- [frame](#)
Verweist auf die Ausführungsumgebung der derzeit aktiven Methode.
- [vars](#)
Ein Zeiger auf die erste lokale Variable der aktuellen Methode.

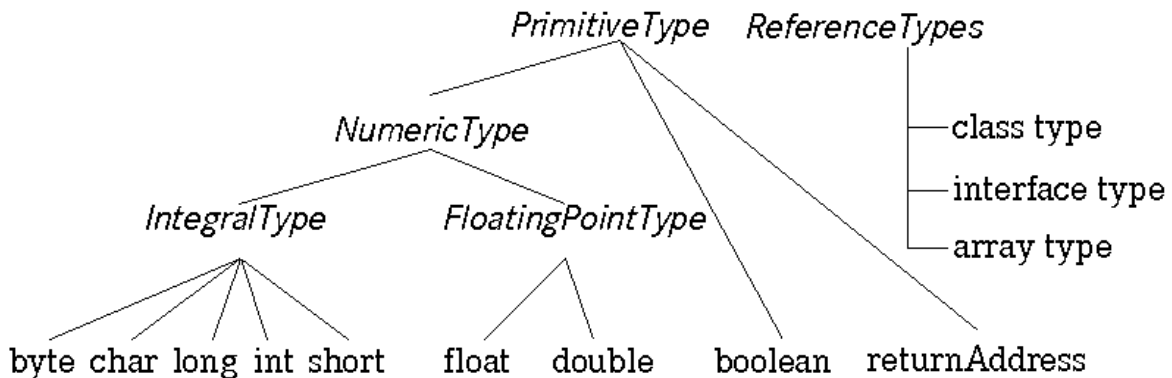
Die Adresslänge innerhalb der JVM ist auf vier Byte (32 Bit) fixiert. Hieraus ergibt sich ein (theoretischer) Adressraum von 4 GB.

Die initiale und maximale Ausdehnung des Heaps kann durch die Kommandozeilenschalter Xms bzw. Xmx gesteuert werden (Beispiel: `java -Xms350M -Xmx500M HelloWorld` führt ein einfaches [Hello-World-Beispiel](#) mit einer

anfänglichen Speicherausstattung von 350 MB aus, die im Verlaufe des Programmablaufs auf höchstens 500 MB anwachsen kann.)

Das **Typsystem der JVM** lehnt sich eng an das der Hochsprache an. ([Zur Erinnerung: primitive Typen in Java](#))

Zusätzlich erweitert es die Primitivtypen um einen Adresstypen `returnAddress` und führt explizite Referenztypen auf die verschiedenen high-level Typen (Klassen, Schnittstellen, Arrays) ein.



Datentypen der JVM:

- `byte`
Vorzeichenbehaftete 8-Bit Zahl in Zweierkomplementdarstellung.
- `short`
Vorzeichenbehaftete 16-Bit Zahl in Zweierkomplementdarstellung.
- `int`
Vorzeichenbehaftete 32-Bit Zahl in Zweierkomplementdarstellung.
- `long`
Vorzeichenbehaftete 64-Bit Zahl in Zweierkomplementdarstellung.
- `char`
Vorzeichenbehaftete 16-Bit Zahl zur Darstellung von Unicode-Zeichen.
- `float`
32-Bit Binärdarstellung einer Fließkommazahl gemäß [IEEE 754-1985](#).
- `double`
64-Bit Binärdarstellung einer Fließkommazahl gemäß [IEEE 754-1985](#).
- `boolean`
Wird zwar durch die JVM definiert, jedoch existieren keine Opcodes, die Parameter dieses Typs erwarten. Der Compiler setzt Anweisungen die `boolean`-Typen enthalten als Operationen auf `int`-Typen um.
- `returnAddress`
Für den Programmierer nicht zugänglicher Typ, der Sprungadressen aufnimmt.
- Referenztypen
Verweisen auf Klassen, Arrays oder Schnittstellen.
Der Wert kann auch `null` sein, wobei die JVM keine konkrete Darstellung dieses Wertes unterstellt.

Jede Bytecode-Instruktion besteht zunächst aus ihrem Opcode, optional gefolgt von den benötigten Operanden. Diese stehen jedoch nicht für sich, sondern sind eingebettet in den organisatorischen Rahmen der `class`-Datei, deren [Format](#) im Anschluß vorgestellt wird.

Die verschiedenen Maschineninstruktionen lassen sich in Klassen einteilen:

Instruktionen zum Zugriff auf lokale Variablen:

Instruktion	Funktion
ret	Rücksprung aus Subroutine, wird in der Implementierung von finally benutzt
aload	Lädt Referenz von spezifisch indizierter Position auf den Operanden-Stack
aload <n>	Lädt Referenz auf Operanden-Stack (Index im Opcode explizit angegeben)
astore	Speichert auf Operanden Stack liegenden Wert in lokale Variable
astore <n>	Legt Referenz in lokaler Variable auf Operanden-Stack ab (Index wird im Opcode explizit angegeben)
dload	Lädt double-Wert aus lokaler Variable auf den Operanden-Stack
dstore	Lädt double-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab
fload	Lädt float-Wert aus lokaler Variable auf den Operanden-Stack
fstore	Lädt float-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab
iload	Lädt int auf Operanden-Stack (Index im Opcode explizit angegeben)
iload <n>	Lädt int-Wert aus lokaler Variable auf den Operanden-Stack
istore	Legt int-Wert von spezifisch indizierter Position in lokaler Variable auf Operanden-Stack ab
istore <n>	Lädt int-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab

<u>lload</u>	Lädt long-Wert aus lokaler Variable auf den Operanden-Stack
<u>lstore</u>	Lädt long-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab
<u>iinc</u>	Inkrementiert lokale Variable um fixe int-Zahl

Instruktionen zur expliziten Modifikation des Operanden-Stacks:

Instruktion Funktion

<u>bipush</u>	Ablegen eines <u>byte-Wertes</u> auf dem Operanden-Stack
<u>sipush</u>	Ablegen eines <u>short-Wertes</u> auf dem Operanden-Stack
<u>pop</u>	Entnimmt und verwirft (de facto: löscht) obersten Operanden-Stack-Eintrag.
<u>pop2</u>	Entnimmt (de facto: löscht) obersten beiden Operanden-Stack-Einträge.
<u>swap</u>	Tauscht die beiden obersten Operanden-Stack-Einträge aus.
<u>ldc</u>	Ablegen eines Elements (referenziert über 16-Bit Index) des <u>runtime constant pools</u> auf dem Operanden-Stack
<u>ldc_w</u>	Ablegen eines Elements (referenziert über 32-Bit Index) des <u>runtime constant pools</u> auf dem Operanden-Stack
<u>aconst_null</u>	Legt null auf dem Operanden-Stack ab.
<u>dconst_<d></u>	Legt double Konstante auf dem Operanden-Stack ab. dconst_0 die Konstante 0.0, bzw. dconst_1 den Wert 1.0.
<u>fconst_<f></u>	Legt float Konstante auf dem Operanden-Stack ab. fconst_0 die Konstante 0.0, bzw. fconst_1 den Wert 1.0.
<u>iconst_<i></u>	Legt int-Konstante auf dem Operanden-Stack ab. Es existieren Opcodes für folgende Konstanten: iconst_m1 -- -1; iconst_0 -- 0; iconst_1 -- 1; iconst_2 -- 2; iconst_3 -- 3; iconst_4 -- 4; iconst_5 -- 5.
<u>dup</u>	Dupliziert oberstes Element des Operanden-Stacks. <u>dup_x1</u> legt das neue Element als zweitunterstes auf dem Stack ab. <u>dup_x2</u> legt das neue Element, abhängig vom Stack-Inhalt, als zweit- oder drittunterstes auf dem Stack ab.
<u>dup2</u>	Dupliziert die beiden obersten Elemente des Operanden-Stacks. <u>dup2_x1</u> legt die neuen Elemente als zweitunterstes und folgendes auf dem Stack ab. <u>dup2_x2</u> legt die neuen Elemente, abhängig vom Stack-Inhalt, als zweit- oder drittunterstes und folgendes auf dem Stack ab.
<u>lconst_<l></u>	Legt long Konstante auf dem Operanden-Stack ab. Es existieren Opcodes für folgende Konstanten: iconst_0 -- 0; iconst_1 -- 1.

Instruktion zur Steuerung des Kontrollflusses:

Instruktion Funktion

<u>goto</u>	Bedingungsloser Sprung innerhalb derselben Methode. Der anzugebende <i>branch offset</i> ist auf 16-Bit fixiert, woraus sich ableiten läßt, daß das Opcodesegment einer Methode niemals (in der JVM-Version 1.3) die Größe von 64KByte überschreiten darf. Näheres zu den Einschränkungen der virtuellen Maschine findet sich in im <u>Abschnitt 4.10 der JVM-Spezifikation</u> , sowie in der <u>Diskussion des Class-File Formats</u> .
<u>goto_w</u>	Bedingungsloser Sprung innerhalb derselben Methode (mit 32-Bit Offset)
<u>if_acmp<cond></u>	Bedingter Sprung im Falle der Gültigkeit der Bedingung. Die zu vergleichenden Operanden werden als Referenzen übergeben. Als Bedingungen stehen Gleichheit (eq) und Ungleichheit (ne) zur Verfügung.
<u>if_icmp<cond></u>	Bedingter Sprung im Falle der Gültigkeit der Bedingung. Die zu vergleichenden Operanden werden als int-Werte übergeben. Als Bedingungen stehen zur Verfügung: Gleichheit (eq), Ungleichheit (ne), Kleiner (lt), Kleiner oder Gleich (le), Größer (gt) und Größer oder Gleich (ge).
<u>if<cond></u>	Bedingter Sprung, nach Vergleich des obersten Elements des Operanden-Stacks mit Null Für spezifische Vergleiche stehen folgende Opcodes zur Verfügung: Geleicheit (ifeq), Ungleichheit (ifne), Kleiner (iflt), Kleiner oder Gleich (ifle), Größer (ifgt), Größer oder Gleich (ifge).
<u>ifnonnull</u>	Bedingter Sprung -- im Falle der Ungleichheit -- nach Vergleich der auf dem Operanden-Stack befindlichen Referenz mit Null
<u>ifnull</u>	Bedingter Sprung -- im Falle der Gleichheit -- nach Vergleich der auf dem Operanden-Stack befindlichen Referenz mit Null
<u>jsr</u>	Unbedingter Sprung, unter Sicherung der Rücksprungadresse auf dem Operanden-Stack, zu 16-Bit Adresse.
<u>jsr_w</u>	Unbedingter Sprung, unter Sicherung der Rücksprungadresse auf dem Operanden-Stack, zu 32-Bit Adresse.
<u>lookupswitch</u>	Zugriff auf Sprungtabelle per Schlüssel und anschließende Verzweigung. Benutzt zur Implementierung des <u>switch</u> -Konstrukts
<u>tableswitch</u>	Indexbasierter Zugriff auf Sprungtabelle und anschließende Verzweigung.

Instruktionen zur Operation auf Klassen und Objekten:

Instruktion Funktion

[anewarray](#) Array-Erzeugung an definierter Stelle im [Laufzeit-Konstanten-Pool](#).

[checkcast](#) Typkompatibilitätsprüfung ([siehe Beispiel](#))

[instanceof](#) Prüft ob Objekt gegebenen Typ hat (d.h. Ausprägung der Klasse -- oder einer Subklasse -- ist; das Interface implementiert; Array-Kompatibel ist)

[new](#) Erzeugt neues Objekt
Hinweis: Die Punktnotation zur Trennung der Pakethierarchien wird hier durch Slash „/“ ersetzt.

Instruktionen zur Methodenausführung:

Instruktion	Funktion
invokespecial	Ruft Instanzenmethode auf; mit besonderer Behandlung bestimmter Umstände. <i>Hinweis:</i> Diese Instruktion wurde umbenannt, frühere JDK-Versionen benutzen invokeonvirtual.
invokestatic	Ruft statische Klassenmethode auf.
invokevirtual	Ruft Instanzenmethode auf.
invokeinterface	Ruft Schnittstellenmethode auf.
areturn	Retourniert Referenz auf Speicherobjekt nach Methodenausführung.
dreturn	Retourniert double-Wert nach Methodenausführung.
freturn	Retourniert float-Wert nach Methodenausführung.
ireturn	Retourniert int-Wert nach Methodenausführung.
lreturn	Retourniert long-Wert nach Methodenausführung.
return	Retourniert nach Methodenausführung ohne Rückgabewert (void-Methode).

Instruktionen zum Zugriff auf Attribute:**Instruktion Funktion**

	Legt Attributinhalt eines Objekts auf Operanden-Stack ab.
getfield	Die Klasse des Objekts, auf das der Zugriff erfolgen soll, wird über einen 16-Bit Offset auf dem runtime constant pool adressiert. Auch hier wird wieder die Limitierung der virtuellen Maschine auf 2 ¹⁶ Klassen deutlich.
getstatic	Legt Attributinhalt eines statischen Klassenattributes auf dem Operanden-Stack ab.
putfield	Setzt Wert eines Attributs.
putstatic	Setzt Wert eines statischen Klassenattributes.

Instruktionen zur Operation auf Arrays:

Instruktion	Funktion
newarray	Erzeugt einen neuen Array, und definiert die Komponententypen als einen der Primitivtypen . Die Implementierung von SUN verwendet für den Wahrheitswert je acht Bit. Anderen Umsetzungen ist es explizit Freigestellt hier mit optimierteren Speicherstrukturen zu operieren.
anewarray	Erzeugt einen neuen Array von Referenztypen zur Aufnahme beliebiger Objekte.
multianewarray	Erzeugt einen mehrdimensionalen Array.
aaload	Lädt Referenz von Arrayposition.
aastore	Speicher Objekt an spezifischer Arrayposition.
arraylength	Liefert Elementanzahl (Kardinalität) eines Arrays.
baload	Lädt byte oder boolean aus Arrayposition.
bastore	Legt byte oder boolean an Arrayposition ab.
saload	Lädt short aus Arrayposition.
sastore	Legt short an Arrayposition ab.
caload	Lädt char aus Arrayposition.
castore	Legt char an Arrayposition ab.
daload	Lädt double aus Arrayposition.

<u>dastore</u>	Legt double an Arrayposition ab.
<u>faload</u>	Lädt float aus Arrayposition.
<u>fastore</u>	Legt float an Arrayposition ab.
<u>iaload</u>	Lädt int aus Arrayposition.
<u>iastore</u>	Legt int an Arrayposition ab.
<u>laload</u>	Lädt long aus Arrayposition.
<u>lastore</u>	Legt long an Arrayposition ab.

Instruktionen zur Typkonversion:

Instruktion	Funktion
<u>i2b</u>	Konvertiert int zu byte.
<u>i2c</u>	Konvertiert int zu char.
<u>i2d</u>	Konvertiert int zu double.
<u>i2f</u>	Konvertiert int zu float.
<u>i2l</u>	Konvertiert int zu long.
<u>i2s</u>	Konvertiert int zu short.
<u>d2f</u>	Konvertiert double zu float.
<u>l2d</u>	Konvertiert long zu double.
<u>l2f</u>	Konvertiert long zu float.
<u>l2i</u>	Konvertiert long zu int.
<u>d2f</u>	Konvertiert double zu float.
<u>f2d</u>	Konvertiert float zu double.
<u>f2i</u>	Konvertiert float zu int.
<u>f2l</u>	Konvertiert float zu long.
<u>d2f</u>	Konvertiert double zu float.
<u>d2f</u>	Konvertiert double zu float.
<u>d2i</u>	Konvertiert double zu int.
<u>d2l</u>	Konvertiert double zu long.

Instruktionen zur Durchführung arithmetischer Operationen:

Eingangssperanden werden vom Stack entnommen und das Berechnungsergebnis ebenda abgelegt.

Instruktion	Funktion
<u>dadd</u>	Addiert zwei double-Werte.
<u>dsub</u>	Subtrahiert zwei double-Werte.
<u>ddiv</u>	Dividiert zwei double-Werte.
<u>dmul</u>	Multipliziert zwei double-Werte.
<u>dneg</u>	Negiert double-Wert durch Zweierkomplementbildung.
<u>drem</u>	Divisionsrest bei Division zweier double-Werte.
<u>dcmp<op></u>	Vergleicht zwei double Werte. Ist einer der beiden Operanden NaN, so legt dcmpg1, dcmpl hingegen -1 als Ergebnis auf dem Stack ab.
<u>fadd</u>	Addiert zwei float-Werte.
<u>fsub</u>	Subtrahiert zwei float-Werte.
<u>fmul</u>	Multipliziert zwei float-Werte.
<u>fdiv</u>	Dividiert zwei float-Werte.
<u>iadd</u>	Addiert zwei int-Werte.
<u>isub</u>	Subtrahiert zwei int-Werte.
<u>imul</u>	Multipliziert zwei int-Werte.

<u>idiv</u>	Dividiert zwei <code>int</code> -Werte.
<u>iand</u>	Boole'sche UND-Verknüpfung zweier <code>int</code> -Werte.
<u>ior</u>	Boole'sche ODER-Verknüpfung zweier <code>int</code> -Werte.
<u>ixor</u>	Exklusive Boole'sche ODER-Verknüpfung zweier <code>int</code> -Werte.
<u>ineg</u>	Negiert <code>int</code> -Wert durch Zweierkomplementbildung.
<u>irem</u>	Divisionsrest bei Division zweier <code>int</code> -Werte.
<u>ishl</u>	Linksshift eines <code>int</code> -Wertes.
<u>ishr</u>	Rechtsshift eines <code>int</code> -Wertes.
<u>iushr</u>	Rechtsshift eines <code>int</code> -Wertes unter Nulleinfügung und Vorzeichenlösung.
<u>fneg</u>	Negiert <code>float</code> -Wert durch Zweierkomplementbildung.
<u>frem</u>	Divisionsrest bei Division zweier <code>float</code> -Werte.
<u>fcmp<op></u>	Vergleicht zwei <code>float</code> Werte. Ist einer der beiden Operanden NaN, so legt <code>fcmpg1</code> , <code>fcmpl</code> hingegen <code>-1</code> als Ergebnis auf dem Stack ab.
<u>ladd</u>	Addiert zwei <code>long</code> -Werte.
<u>land</u>	Boole'sche UND-Verknüpfung zweier <code>long</code> -Werte.
<u>ldcmp</u>	Vergleicht zwei <code>long</code> Werte.
<u>ldiv</u>	Dividiert zwei <code>long</code> -Werte.
<u>lmul</u>	Multipliziert zwei <code>long</code> -Werte.
<u>lneg</u>	Negiert <code>long</code> -Wert durch Zweierkomplementbildung.
<u>lrem</u>	Divisionsrest bei Division zweier <code>long</code> -Werte.
<u>lor</u>	Boole'sche ODER-Verknüpfung zweier <code>long</code> -Werte.
<u>lshl</u>	Linksshift eines <code>long</code> -Wertes.
<u>lsht</u>	Rechtsshift eines <code>long</code> -Wertes.
<u>lsub</u>	Subtrahiert zwei <code>long</code> -Werte.
<u>lushr</u>	Rechtsshift eines <code>long</code> -Wertes unter Nulleinfügung und Vorzeichenlösung.
<u>lxor</u>	Exklusive Boole'sche ODER-Verknüpfung zweier <code>long</code> -Werte.

Sonstige Instruktionen:

Instruktion	Funktion	
Ausnahmebehandlung	<u>athrow</u>	Wirft eine Exception.
Synchronisation -- Monitoroperationen	<u>monitorenter</u>	Der Zugriff auf jedes Objekt wird durch einen Monitor synchronisiert. Die Anweisung sperrt ein Objekt.
	<u>monitorexit</u>	Gibt ein gesperrtes Objekt frei.
Sonstige Opcodes	<u>nop</u>	Buchstäblich: <i>no operation</i> . Bewirkt nichts; keine Operatoren, keine Änderungen am Operanden-Stack.

Die beiden Opcodes mit den Ordnungsnummern 254 und 255 (0xfe und 0xff, mnemonic `impdep1` und `impdep2`) sind durch SUN als reserviert gekennzeichnet. Sie können von durch den Hersteller der virtuellen Maschine mit eigendefinierter Funktionalität implementiert werden.

Darüberhinaus ist mit dem Opcode 202 (mnemonic `breakpoint`) ein Einstiegspunkt für Debugger definiert.

Alle Opcodes sind mit einem Byte codiert. Hieraus ergibt 256 als maximaler Befehlsumfang der virtuellen Maschine. Zur Verringerung der notwendigen verschiedenen Befehle sind nicht alle Opcodes für alle [Typen der JVM](#) implementiert. Üblicherweise existieren nur Opcodes für `int`, `float`, `long` und `double` sowie die Referenzen. Für alle anderen Typen stehen Konvertierungsmöglichkeiten in die genannten zur Verfügung.

Opcode	byte	short	int	long	float	double	char	Referenz
<code>...ipush</code>	<code>bipush</code>	<code>sipush</code>						
<code>...const</code>			<code>iconst</code>	<code>lconst</code>	<code>fconst</code>	<code>dconst</code>		<code>aconst</code>
<code>...load</code>			<code>iload</code>	<code>lload</code>	<code>fload</code>	<code>dload</code>		<code>aload</code>
<code>...store</code>			<code>istore</code>	<code>lstore</code>	<code>fstore</code>	<code>dstore</code>		<code>astore</code>
<code>...inc</code>			<code>inc</code>					

...aload	baload	saload	iaload	laload	faload	daload	caload	aaload
...astore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
...add			iadd	ladd	fadd	dadd		
...sub			isub	lsub	fsub	dsub		
...mul			imul	lmul	fmul	dmul		
...div			idiv	ldiv	fdiv	ddiv		
...rem			irem	lrem	frem	drem		
...neg			ineg	lneg	fneg	dneg		
...shl			ishl	lshl				
...shr			ishr	lshr				
...ushr			iushr	lushr				
...and			iand	land				
...or			ior	lor				
...xor			ixor	lxor				
i2...	i2b	i2s		i2l	i2f	i2d		
l2...			l2i		l2f	l2d		
f2...			f2i	f2l	f2d			
...cmp				lcmp				
...cmpl				fcmpl	dcmpl			
...cmpg				fcmpg	dcmpg			
if_...cmpcond			if_icmpcond					if_acmpcond
...return			ireturn	lreturn	freturn	dreturn		areturn

Treten bei der Ausführung der Opcodes Ausnahmen auf, so werden durch die virtuelle Maschine [Laufzeit-Ausnahmeereignisse](#) (engl. *runtime exception*) generiert. [Wie bekannt](#) werden Ausnahmen dieser Kategorie (üblicherweise) nicht aufgefangen und behandelt. So wird die [ClassCastException](#) im [Beispiel aus Kapitel 2](#) durch die versuchte explizite Typumwandlung ausgelöst. Der erzeugte Bytecode für diese Anweisung lautet:

```
aload_3
checkcast 2
```

aload_3 lädt die Referenz auf eine lokale Variable auf den Operanden-Stack. Die lokale Variable 3 entspricht im Beispiel myC11. checkcast testet ob die auf dem Operanden-Stack befindliche Referenz kompatibel zum übergebenen Typen (hier die 2 als Referenz auf die zweite geladene Klasse; benannt mit C2) ist. Im Falle der Inkompatibilität wird durch die virtuelle Maschine eine ClassCastException erzeugt.

Beispiel 6: Einfache arithmetische und Ein-/Ausgabeoperationen

```
(1).class public examples/BC1
(2).super java/lang/Object
(3)
(4).method public <init>()V
(5)  aload_0
(6)  invokevirtual java/lang/Object/<init>()V
(7)  return
(8).end method
(9)
(10).method public static main([Ljava/lang/String;)V
(11)  .limit locals 5
(12)  .limit stack 10
(13)
(14)  iconst_2
(15)  istore_0
(16)
(17)  bipush 101
(18)  istore_1
(19)
(20)  bipush 99
(21)  istore_2
(22)
(23)  ;we will need this twice
(24)  getstatic java/lang/System/out Ljava/io/PrintStream;
(25)  astore_3
(26)
(27)  iload_1
```




```

(28)    iload_2
(29)    iadd
(30)    istore_1
(31)
(32)    ;convert int to string
(33)    iload_1
(34)    invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
(35)    astore 4
(36)
(37)    ;Print a string
(38)    aload_3
(39)    aload 4
(40)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(41)
(42)    iload_1
(43)    iload_0
(44)    idiv
(45)
(46)    ;convert int to string
(47)    invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
(48)    astore 4
(49)
(50)    ;Print a string
(51)    aload_3
(52)    aload 4
(53)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(54)
(55)    ;print a fixed string
(56)    aload_3
(57)    ldc "The End"
(58)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(59)    return
(60).end method
(61)

```

[Download des Beispiels](#)

Der Java-Assemblercode des Beispiels 6 zeigt die Verwendung einiger einfacher arithmetischer und Ein-/Ausgabeoperationen.

Zunächst zeigt das Beispiel den Aufbau einer Java-Assemblerdatei, wie sie vom Übersetzer [Jasmin](#) akzeptiert und in ausführbaren Java-Bytecode umgewandelt wird.

So legt die `.class`-Deklaration zunächst fest, daß es sich um die öffentlich zugängliche (d.h. als `public` deklarierte) Klasse `BC1`, im Paket `examples` handelt.

Die darauf folgende `.super`-Definition legt die Elternklasse der betrachteten Klasse fest. Im Falle keiner explizit definierten Elternklasse ist dies vorgabegemäß die Standardklasse [Class](#).

Nach den einführenden Deklarationen definiert die Quellcodedatei den statischen Initialisierer.

Die Methode `main` stellt den Beginn der aktiven Verarbeitung dar. Ihre Signaturdefinition `([Ljava/lang/String;)V` läßt die JVM-interne Kurzschreibweise des Typsystems erkennen. So deutet die in den Klammern der Übergabewerte eingeschlossene eckige Klammern an, daß ein Array gleichtypisierter Ausprägungen übergeben wird. Diese Ausprägungen sind alle vom Standardtyp `java.lang.String` (die paket-separierenden Punkte werden JVM-intern zu Pfadseparatoren aufgelöst). Zusätzlich ist dem Klassennamen ein einleitendes `L` vorangestellt, um auszudrücken, daß es sich nicht um einen Primitivtyp, sondern um eine Sprachkomponente (das „`L`“ deutet hierbei auf den Begriff *language* hin) handelt.

Nach der Klammer ist der Rückgabetyt --- im Falle von `main` vorgabegemäß `void` --- angegeben. Auch er wird unter Verwendung derselben Abkürzungskonvention dargestellt.

Zu Eingangs der Methode `main` allozieren die beiden Direktiven `.limit` zunächst Speicher für die lokalen Variablen (`.limit locals`) und die Tiefe des methodenintern verwendeten Operandenstacks (`.limit stack`).

Die (aktive durch den Programmierer gesteuerte) Verarbeitung beginnt im Beispiel mit dem Anweisung `iconst_2` welche den ganzzahligen Wert 2 auf dem Operandenstack ablegt. Anschließend wird dieser Wert, mittels der Anweisung `istore_0` vom Stack entnommen und in die erste lokale Variable gespeichert.

Mit [iconst_2](#) findet ein besonderer Befehl zur Ablage einer ganzzahligen Konstante auf dem Operanden Stack Verwendung, der es gestattet bestimmte (häufig benötigte) Konstantenablagen in nur genau einem Byte auszudrücken. Durch die JVM-Spezifikation vorgesehen sind hierbei Instruktionen für die Konstanten -1, 0, 1, 2, 3, 4 oder 5. Im Ergebnis ist die Nutzung der abkürzenden Befehlsschreibweise äquivalent zum Einsatz der Instruktion [bipush](#) unter Explizierung der abzulegenden Konstante.

Diese äquivalente Form der Belegung einer lokalen Variable zeigt der zweite Anweisungsblock, der die numerische Konstante 101, für die keine abkürzende Schreibweise angeboten wird, auf dem Operandenstack ablegt um sie der zweiten lokalen Variable (mit der Indexnummer 1) zuzuweisen.

In derselben Weise wird für die Initialisierung der dritten lokalen Variablen mit dem Wert 99 verfahren.

Anschließend wird durch `getstatic` der Dateideskriptor der Standardausgabe (d.h. desjenigen Streams mit dem Wert `System/out`) gelesen und die zurückgelieferte Adresse in der vierten lokalen Variable (Indexnummer 3) abgelegt.

Der darauffolgende Anweisungsblock zeigt die Umsetzung einer einfachen Ganzzahladdition, die zunächst die beiden zu verknüpfenden Operanden (die Inhalte der lokalen Variablen mit den Indexnummern 1 und 2) auf dem Stack

ablegt und anschließend mittels der Ganzzahladdition (`iadd`) verknüpft.

Das auf dem Stack abgelegte Berechnungsergebnis wird durch `istore_1` in die zweite lokalen Variablen zugewiesen.

Der nächste Anweisungsblock bereitet die Ausgabe des Berechnungsergebnisses auf der Standardausgabe vor. Hierzu platziert er zunächst den Inhalt der lokalen Variable mit der Indexnummer 1 (d.h. das Berechnungsergebnis des direkt vorhergehenden Schrittes) auf dem Stack.

Anschließend wird eine Standard-API-Methode (die Methode `valueOf`) aufgerufen, welche den auf dem Stack übergebenen `int`-Parameter in eine Zeichenkette wandelt und die Referenz darauf als Rückgabewert auf dem Stack platziert.

Dieser Rückgabewert wird in der fünften lokalen Variable (Indexnummer 4) abgelegt.

Anschließend werden die in zwischenzeitlich den vierten und fünften lokalen Variablen abgelegten Adressen des Ausgabe-Streams und der auszugebenden Zeichenkette geladen und auf dem Operanden-Stack abgelegt.

Durch Aufruf der Standard-Ausgabemethode `println` mittels `invokevirtual` wird die referenzierte Zeichenkette auf der Standardausgabe dargestellt.

Der folgende Anweisungsblock demonstriert eine Ganzzahldivision mittels `idiv` welche Divisor und Dividenden als Operanden auf dem Stack erwartet und das Berechnungsergebnis ebenda platziert.

Anschließend wird das (noch auf dem Stack liegende) Berechnungsergebnis direkt weiterverarbeitet und in eine Zeichenkette gewandelt. Hierbei kommt die bereits bekannte Funktion zum Einsatz.

Danach erfolgt wiederum die Ausgabe in der bekannten Form.

Abschließend wird eine fixe Zeichenkette ausgegeben, deren Zeichenkettendarstellung nicht berechnet zu werden braucht. Ihr Wert kann daher direkt aus dem Laufzeitkonstantenpool per `ldc` geladen werden.

Die übrigen Schritte zur Erzeugung der Ausgabe bleiben indes unverändert.

Nutzung von Methoden:

Bereits bei den einfachen Operationen aus Beispiel 6 zeigt sich, daß die wiederholte Angabe von sehr ähnlichen Instruktionsfolgen nicht zu vermeiden ist. Insbesondere die beiden Konversionen des `int`-Datentyps als Voraussetzung der zeichenbasierten Ausgabe ist vollständig identisch.

Zur Strukturierung stehen daher auf der Java-Assemblerebene die bereits aus der Java-Hochsprache bekannten Methoden zur Verfügung, wie Beispiel 7 zeigt.

Beispiel 7: Nutzun von Methoden

```
(1).class public examples/BC2
(2).super java/lang/Object
(3)
(4).method public <init>()V
(5)    aload_0
(6)    invokevirtual java/lang/Object/<init>()V
(7)    return
(8).end method
(9)
(10).method public static printInt(I)V
(11)    .limit locals 2
(12)    .limit stack 2
(13)
(14)    iload_0
(15)    invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
(16)    astore_1
(17)
(18)    getstatic java/lang/System/out Ljava/io/PrintStream;
(19)    aload_1
(20)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(21)    return
(22).end method
(23)
(24).method public static main([Ljava/lang/String;)V
(25)    .limit locals 50
(26)    .limit stack 40
(27)
(28)    iconst_2
(29)    istore_0
(30)
(31)    bipush 101
(32)    istore_1
(33)
(34)    bipush 99
(35)    istore_2
(36)
(37)    ;we will need this twice
(38)    getstatic java/lang/System/out Ljava/io/PrintStream;
(39)    astore_3
(40)
(41)    iload_1
(42)    iload_2
(43)    iadd
(44)    istore_1
(45)
(46)    iload_1
```



```

(47)    invokestatic  examples/BC2/printInt(I)V
(48)
(49)    iload_1
(50)    iload_0
(51)    idiv
(52)
(53)    invokestatic  examples/BC2/printInt(I)V
(54)
(55)    ;print a fixed string
(56)    aload_3
(57)    ldc "The End"
(58)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(59)    return
(60).end method

```

[Download des Beispiels](#)

Die Funktionalität des Beispiels ist mit der der vorhergehend vorgestellten Codesequenz identisch. Jedoch finden sich jetzt die Instruktionsfolgen zur Berechnung der Zeichenkettenrepräsentation einer Ganzzahl und ihrer anschließenden Ausgabe in die Methode `printInt` ausgelagert.

Diese Methode akzeptiert eine Ausprägung des Primitivtyps `int` als Übergabe und liefert keinen Rückgabewert. Die Signatur ist daher dahingehend vereinbart, daß genau eine `int`-konforme Zahl als Parameter auf dem Stack erwartet wird, d.h. der Aufrufer hat diese vor dem Aufruf dort abzulegen.

Zusätzlich benötigt die Methode selbst zu ihrer Ausführung einige lokale Variablen, die auf dem methodenspezifischen Stack abgelegt werden. Dieser stellt eine Erweiterung des bereits durch den Aufruf verwendeten Operandenstacks dar.

Mit Java steht jedoch keineswegs die einzige Hochsprache zur Erzeugung von Byte-Code-Dateien zur Verfügung.

[Diese Seite](#) listet eine Vielzahl verschiedener Alternativen.

Beispielsweise erzeugt der [Oberon-Compiler von Canterbury](#) für alle Oberon-Module, einschließlich der Systemmodule, Java-Klassen.

Beispiel 8: Die Hello World Applikation als Oberon Programm



```

MODULE helloworld;

IMPORT Out;

BEGIN
  Out.String( "Hello World" );
  Out.Ln;
END helloworld.

```

[Download des Beispiels](#)

Die erzeugten class-Dateien -- [SYSTEM.class](#), [helloworld.class](#), [Out.class](#), [Sys.class](#) -- können auf jeder JVM zur Ausführung gebracht werden. `java helloworld` liefert das erwartete Ergebnis.

Das **Class-File-Format**

Ausgangspunkt jeder Programmausführung innerhalb der JVM ist die class-Datei als Eingabe. Sie wird üblicherweise durch den Java-Compiler (im JDK: [javac](#) erzeugt).

Einige Eigenschaften jedes class-Files:

- Es enthält die Definition genau einer Klasse oder einer Schnittstelle, unabhängig von deren Zugriffs- und Sichtbarkeitseigenschaften.
- Es wird als Bytestrom aufgefaßt, daher werden alle größeren Informationseinheiten durch serielles Lesen und aneinanderfügen gebildet.
Unabhängig von der physischen Realisierung der tatsächlich zugrundeliegenden Hardware werden alle Multibyte-Daten im *big endian*-Format, d.h. größere Einheiten werden durch Linksshift und Boole'sches ODER gebildet, abgelegt. Die Schnittstellen [java.io.DataInput](#), [java.io.DataOutput](#), [java.io.DataInputStream](#) und [java.io.DataOutputStream](#) unterstützen dieses Format. ([Beispiel](#))
- Die Strukturen innerhalb der class-Datei werden nicht zusätzlich optimiert abgelegt, daher erfolgt weder ein Auffüllen auf spezifische Wortgrenzen, noch ein Alignment an solchen.

Die JVM-Spezifikation legt zur Definition der Struktur des class-Files [eigene Datentypen](#) fest: `u1`, `u2` und `u4` zur Definition vorzeichenloser ein-, zwei- und drei-Bytetypen. Für diese (von der Java-üblichen vorzeichenbehafteten Mimik (abgesehen von `char`) abweichenden) Datentypen stehen mit [readUnsignedByte\(\)](#), [readUnsignedShort\(\)](#) und [readInt\(\)](#) entsprechende Lesemethoden zur Verfügung.

[The class File Format @ Java Virtual Machine Specification](#)

```

ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;

```

```

u2 constant_pool_count;
cp_info constant_pool[constant_pool_count-1];
u2 access_flags;
u2 this_class;
u2 super_class;
u2 interfaces_count;
u2 interfaces[interfaces_count];
u2 fields_count;
field_info fields[fields_count];
u2 methods_count;
method_info methods[methods_count];
u2 attributes_count;
attribute_info attributes[attributes_count];
}

```

Constant Pool @ Java Virtual Machine Specification

```

cp_info {
    u1 tag;
    u1 info[];
}

```

field_info @ Java Virtual Machine Specification

```

field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}

```

method_info @ Java Virtual Machine Specification

```

method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}

```

attribute_info @ Java Virtual Machine Specification

```

attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}

```

Aufbau einer class-Datei verdeutlicht an nachfolgendem Beispiel Quellcode.

Hinweis: Mit [Classeditor](#) existiert ein freies Werkzeug zur Inspektion und Modifikation übersetzter Class-Dateien.

Beispiel 9: Java-Quellcode der untersuchten Klassendatei

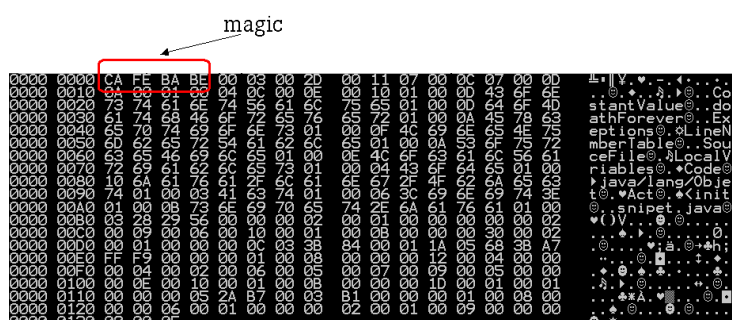
```

(1) class Act {
(2)     public static void doMathForever() {
(3)         int i=0;
(4)         while (true) {
(5)             i += 1;
(6)             i *= 2;
(7)         } //while
(8)     } //doMathForever()
(9)} //class Act

```



Download des Beispiels



Der magic-Identifier ist auf die Bytekombination (in hexadezimaler Darstellung) CA FE BA BE fixiert. Anhand dieser erkennt der Kassenlader der Laufzeitsystems die Datei als ausführbare Java-Bytecode-Datei an. Ist diese gegenüber dem Vorgabewert modifiziert wird eine `java.lang.ClassFormatError` Ausnahme im Hauptthread generiert (Bad magic number wird als zusätzliche Nachricht der Ausnahme ausgegeben). [siehe JVM-Spezifikation](#)



Die beiden Versionskennungen `minor` und `major` bilden gemeinsam den Versionsschlüssel der `class`-Datei in der gängigen Punktnotation. Hierbei gilt: `major.minor`

Die Klassendatei des Beispiels trägt den Versionsschlüssel 45.3.

Der im SUN JDK enthaltene Java-Compiler erlaubt per Kommandozeilenparameter (`target`) die JVM-spezifische Steuerung der Codegenerierung. Die den einzelnen Sprachversionen zugeordneten Bytecodeversionen sind in der nachfolgenden Tabelle zusammengestellt.

Java-Version	Bytecode-Version
1.1	45.3
1.2	46.0
1.3	47.0
1.4 (sowie 1.4.1, 1.4.2 und der Prototyp des 1.5-Compilers)	48.0
Zweite Vorabversion des 1.5-Compilers	50.0
1.5 (beta1)	49.0
1.5 (beta2)	49.29

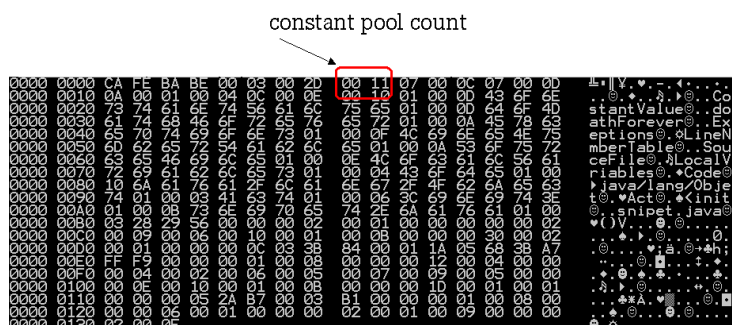
Vorgabegemäß wird durch die Compilerversion 1.5 (ab Beta-Version 2) 49.29 erzeugt.

Trägt eine `class`-Datei eine durch die JVM nicht unterstützte Versionsnummer, so wird eine `java.lang.`

`UnsupportedClassVersionError`-Ausnahme generiert.

Jede JVM kann verschiedene `class`-Datei-Versionen unterstützen, die letztendlich Festlegung welche Versionen durch einzelne Java-Plattform-Releases zu unterstützten sind obliegt jedoch SUN. So unterstützt SUNs JDK v1.0.2 `class`-Dateien der Versionen 45.0 bis einschließlich 45.3. Die JDK-Generation v1.1.x ab Version 45.0 bis einschließlich 45.65535 und Implementierungen der Java 2 Plattform, Version 1.2, bis einschließlich 46.0. JDK v1.3.0 verarbeitet Klassendateien bis hin zur Versionsnummer 47.0. Zur Verarbeitung von Klassen, welche die in 1.5 eingeführten Generizitätsmechanismen verwenden nicht zwingend eine Ausführungsumgebung dieser Versionsnummer benötigt, da das erzeugte Klassenformat (bisher, da diese Aussagen auf dem Informationsstand der verfügbaren Betaversion basieren) nicht verändert wurde. Die Nutzung des dynamischen Boxing/Unboxings benötigt jedoch eine Ausführungsumgebung mindestens der Version 1.5.

[siehe JVM-Spezifikation](#)



Die Bytefolge `constant_pool_count` enthält die um eins erhöhte Anzahl der Einträge der `constant_pool` Tabelle. Im Beispiel ist dies: 17.

[siehe JVM-Spezifikation](#)

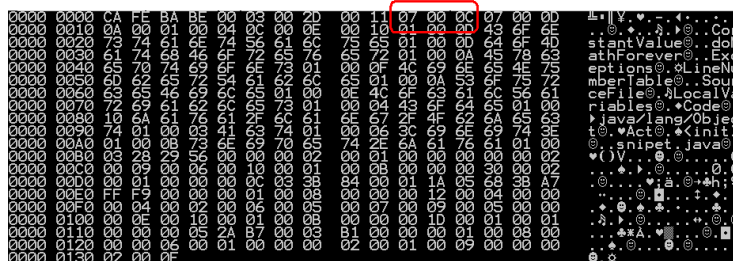
Der `constant pool` enthält die symbolische Information über Klassen, Schnittstellen, Objekte und Arrays. Die Elemente dieser Datenstruktur sind vom Typ `cp_info` und variieren je nach tag in ihrer Länge. Als Konstantentypen (= Inhalt des Tag-Bytes) sind zugelassen:

Konstantentyp**Wert**

CONSTANT_Utf8	1
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_NameAndType	12
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_CLASS	7
CONSTANT_String	8
CONSTANT_Fieldref	9

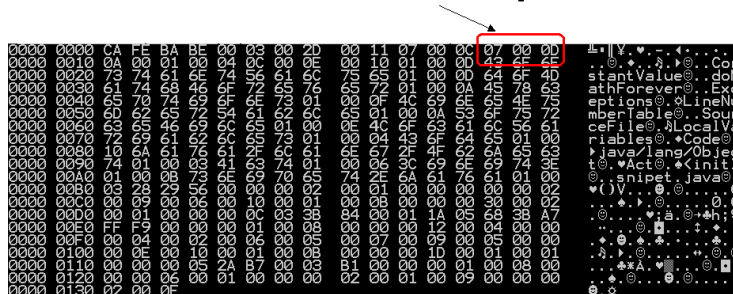
[siehe JVM-Spezifikation](#)

1. Element des constant_pool
(Superklassen-Verweis)



Konstante vom Typ `CONSTANT_class` die einen Verweis auf auf das zwölfte Element des Konstantenpools (0x0C) enthält. Zum Lesezeitpunkt der ersten Konstante kann diese Referenz noch nicht aufgelöst und auf Gültigkeit geprüft werden. (Später sehen wir, daß es sich um eine Referenz auf `java.lang.Object` handelt). Hierbei handelt es sich immer um die Referenz auf die Superklasse. Auch für die API-Klasse `Object` selbst findet sich diese Refenz in der Klassendatei, auch wenn Diassemblierungswerkzeuge wie `javap` diese nicht ausgeben.

2. Element des constant_pool



Konstante vom Typ `CONSTANT_class` die einen Verweis auf auf das 13. Element des Konstantenpools (0x0D) enthält. An dieser Stelle findet sich der String `Act`, also der Klassenname der zur Klassendatei gehörigen Klasse selbst. Auch hierbei handelt es sich zunächst um eine nicht auflösbare Vorwärtsreferenz. Technisch gesehen realisiert sie den `this`-Verweis.

3. Element des constant_pool
(Konstruktorenreferenz)



Konstante vom Typ `CONSTANT_Methodref`. Die folgenden zwei Bytes (im Beispiel: 0x00 01) bezeichnen das referenzierte Objekt, gefolgt vom Methodenindex (0x00 04). Im Beispiel handelt es sich um das Objekt mit der Referenznummer 1 (=erstes Element des Konstantenpools, die Superklasse `Object`). Unter der Referenznummer 0x04 wird auf die Methode `<init>()` verwiesen. Da die betrachtete Klasse `Act` keinen eigenen Konstruktor definiert, wird der der Superklasse aufgerufen.

<http://www.ieckle.de/vorlesung/sysarch/script.html> (31 of 59)08.06.2004 20:56:42

8. Element des constant_pool (Zeichenkette "LineNumberTable")

Konstante vom Typ `CONSTANT_Utf8`, die den fixen String `LineNumberTable` einleitet.

9. Element des constant_pool (Zeichenkette "SourceFile")

Konstante vom Typ `CONSTANT_Utf8`, die den fixen String `SourceFile` einleitet.

10. Element des constant_pool (Zeichenkette "LocalVariable")

Konstante vom Typ `CONSTANT_Utf8`, die den fixen String `LocalVariables` einleitet.

11. Element des constant_pool (Zeichenkette "Code")

Konstante vom Typ `CONSTANT_Utf8`, die den fixen String `Code` einleitet.

12. Element des constant_pool (Zeichenkette "java/lang/Object")

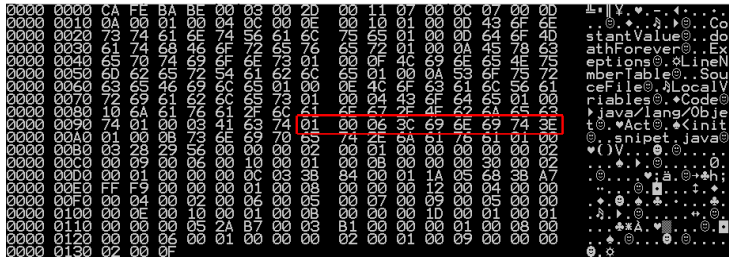
Konstante vom Typ `CONSTANT_Utf8`, die den String `java/lang/Object` einleitet. Vom ersten Element des Konstantenpools referenziertes Element. Die in der Java-Hochsprache üblichen Punkte zur Trennung der Pakete, Subpakete und Klassennamen werden in der JVM konsequent (aus historischen Gründen) durch Querstriche ersetzt.

13. Element des constant_pool (Zeichenkette "Act")



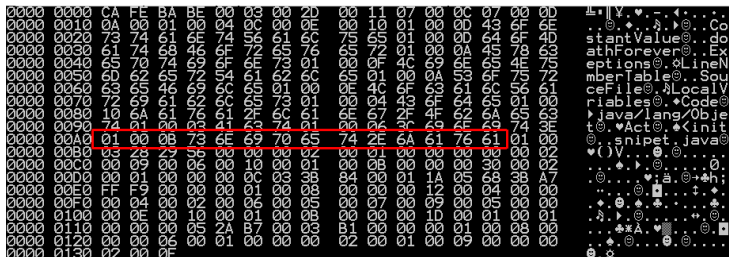
Konstante vom Typ `CONSTANT_Utf8`, die den String `Act` einleitet. Vom zweiten Element des Konstantenpools referenziertes Element. Name der Klasse.

14. Element des constant_pool (Zeichenkette "<init>")



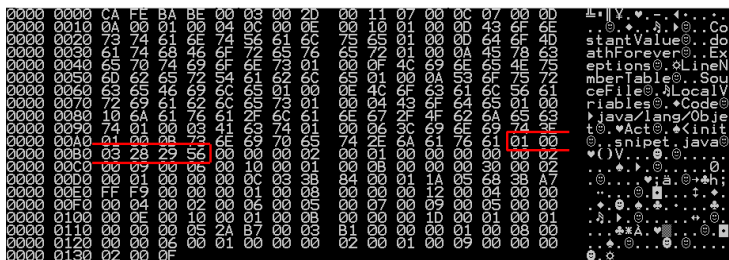
Konstante vom Typ `CONSTANT_Utf8`, die den String `<init>` einleitet. Methodenname der innerhalb der Superklasse `Object`. Der Rückgabewert dieser Methode ist im vierten Element des Konstantenpools abgelegt.

15. Element des constant_pool (Zeichenkette "snipet.java")



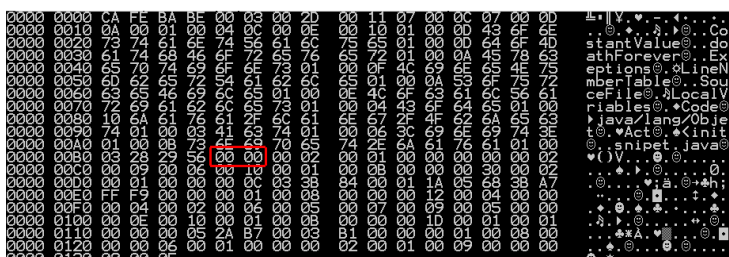
Konstante vom Typ `CONSTANT_Utf8`, die den String `snipet.java` -- den Namen der Quellcodedatei in der sich die Definition der Klasse `Act` befindet -- einleitet.

16. Element des constant_pool (Zeichenkette "()V")



Konstante vom Typ `CONSTANT_Utf8`, die den String `()V` einleitet. Methodendeskriptor, der weder Übergabeargumente noch Rückgabetyt besitzt.

Zugriffsflags



Zugriffsflags für die Klasse `Act`. Der konkrete Code ergibt sich aus der binären ODER-Verknüpfung verschiedener Zugriffsflags, die in untenstehender Tabelle wiedergegeben sind.

[illegible]

```
interface_count und
fields_count
```

[illegible]

methods_count

[illegible]

```
methods_info
(doMathForever())
```

[illegible]

<http://www.ieckle.de/vorlesung/sysarch/script.html> (34 of 59)08.06.2004 20:56:42

[illegible]

Der abschließende vier-Byte Indikator enthält die Länge der Methodenimplementierung (im Beispiel: 0x30).

[illegible]

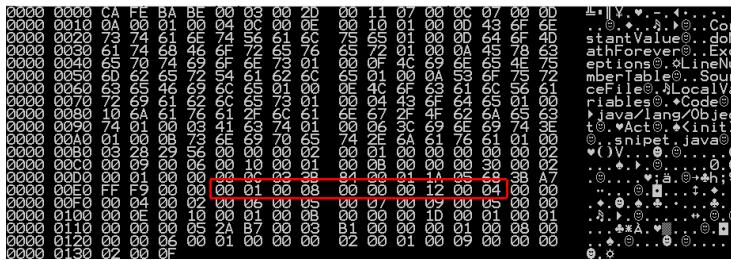
maxLocals: Anzahl der lokalen Variablen. (die verwendete Variable i)

[illegible]

pc	instruction	mnemonic
0	03	iconst_0
1	3B	istore_0
2	840001	ilnc_0 1
5	1A	iload_0
6	05	iconst_2
7	68	imul
8	3B	istore_0
9	A7FFF9	goto 2

<http://www.ieckle.de/vorlesung/sysarch/script.html> (35 of 59)08.06.2004 20:56:42

Eigenschaften des Code-Bereichs der Methode doMathForever()



In diesem Bereich werden zusätzliche Charakteristika des bereits definierten Codebereichs hinterlegt, z.B.

Debugginginformation.

Im betrachteten Falle ist nur eine Eigenschaft angegeben (attribute_count = 0x01). Diese referenziert das achte Element des Konstantenpools -- die Zeichenkette LineNumberTable. Die beiden abschließenden Attribute bezeichnen die Länge dieser Tabelle (0x12) und die Anzahl der Einträge (0x4).

Die LineNumberTable des Beispiels:

```
line 4: i = 0;
line 5: while(true) {
line 6: i += 1;
line 7: i *= 2;
```

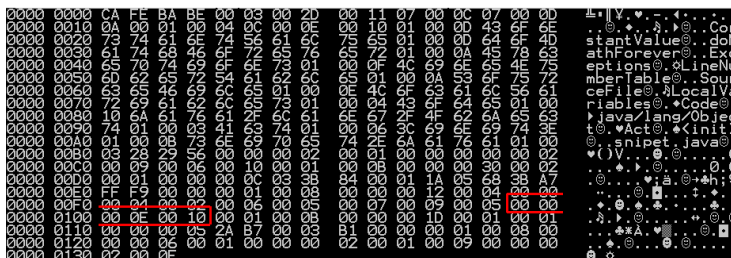
Zuordnung zwischen LineNumberTable und der Methode doMathForever()



Diese Datenstruktur stellt die Zuordnung zwischen den Quellcodezeilen und den resultierenden Opcodes her.

```
LineNumberTable[0]:  iconst_0    istore_0
LineNumberTable[1]:  iinc 0 1
LineNumberTable[2]:  iload_0     iconst_2    imul   istore_0
LineNumberTable[3]:  goto 2
```

Zugriffsflaggen und Indizes der Klasse Act()



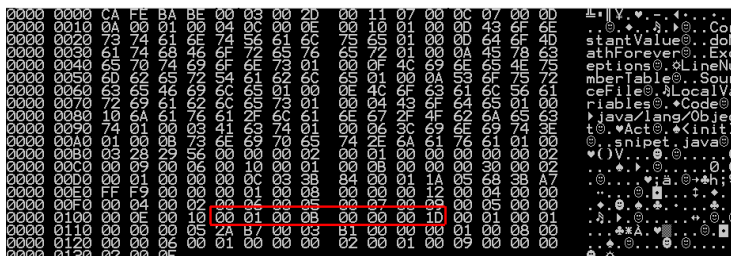
Diese zweite methodenbezogene Struktur gibt Auskunft über den Konstruktor der Klasse Act.

Im ersten Doppelbyte sind die Zugriffsrechte spezifiziert; in diesem Falle sind keine gesonderten Festlegungen getroffen -- es handelt sich um eine *einfache* Methode.

Die Referenz in den Konstantenpool verweist auf die implementierende Methode (im Beispiel: Position 0x0E, dort findet sich die Methode <init>).

Durch die letzten beiden Bytes wird der Typ des Konstruktors referenziert, im betrachteten Beispiel die Position 0x10 im Konstantenpool, mithin ein parameterloser Konstruktor.

Attribute der Klasse Act()



Der Zähler (ersten beiden Bytes) zu Beginn der Struktur zeigt an, daß nur ein Attribut der Klasse `Act()` folgt. Im Beispielfall handelt es sich dabei um das über den Index 11 (0x0B) angesprochene Element des Konstantenpools, die Zeichenkette `Code`.

Der abschließend angegebene Längenzähler fixiert die Anzahl der folgenden Bytes.

maxStack und maxLocals der Klasse

`Act()`

Analog der Definition für Methoden, die maximale Höhe des Operandenstacks und die Anzahl der lokalen Variablen.

Bytecode und Ausnahmetabelle der Klasse

`Act()`

Opcod-Implementierung des Konstruktors, sowie die Aufzählung der durch ihn potentiell ausgelösten Ausnahmereignisse (im keine, daher Anzahl gleich Null).

Die Implementierung in Java-Bytecode:

pc	instruction	mnemonic
0	2A	aload_0
1	B70003	invokeonvirtual #3 <Method java.lang.Object <init> ()V>
4	B1	return

Eigenschaften des Code-Bereichs der Klasse

`Act()`

Anzahl der Eigenschaften im ersten Doppelbyte (im Beispiel: 1). Die spezifische Eigenschaft wird durch Index acht im Konstantenpool (=LineNumberTable) näher definiert.

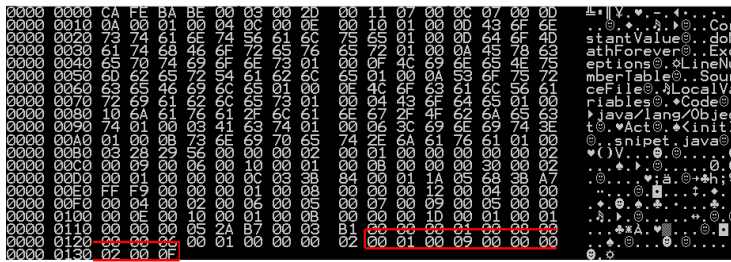
Diese Tabelle hat die Länge 0x06, mit einem einzigen Eintrag.

Zuordnung zwischen LineNumberTable und der Klasse

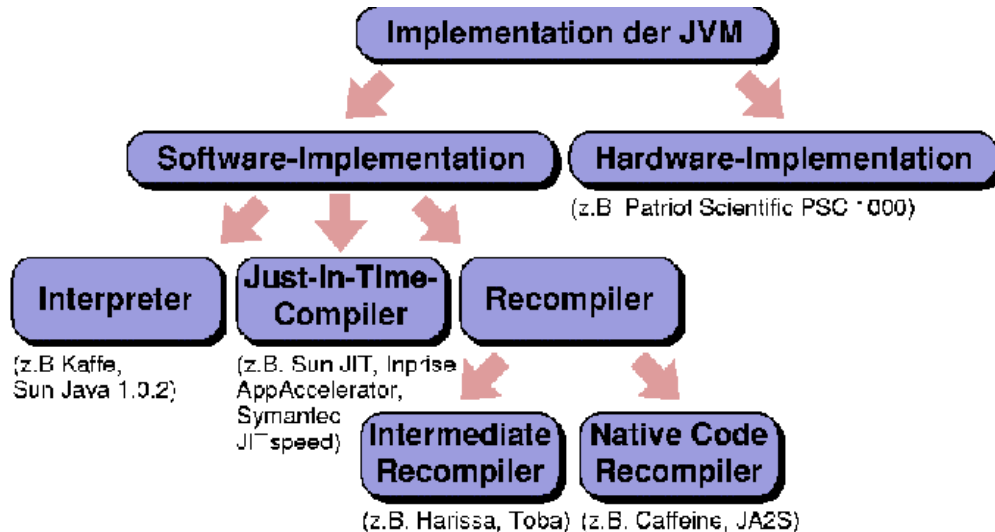
`Act()`

Zuordnung der Quellcodezeilennummern zu den resultierenden Opcodes.

Allgemeine Eigenschaften



Am Ende einer Klassendatei kann eine beliebige Menge allgemeiner Attribute angegeben werden. für das Beispiel wurde durch den Compiler genau ein Attribut erzeugt, ein Verweis auf die Quelldatei (Konstantenpool-Index 0x09). Auf diese Information folgt ein Verweis auf den Namen der Quellcodedatei (Konstantenpool-Index 0x15).

Schlußbemerkungen:

Graphik aus: [Raner, M.: Blick unter die Motorehaube](#)

Die interpretative Ausführung einer class-Datei mit der virtuellen Java-Maschine ist jedoch keineswegs zwingend, auch wenn sie das derzeit am häufigsten anzutreffende Vorgehen verkörpert. Bereits in der Standardedition des Java Development Toolkits von SUN wird seit Version 1.2 ein *just in time compiler* mitgeliefert, der transparent in die Ausführungsumgebung integriert ist. Er durchbricht die befehlswise interpretative Abarbeitung und greift den bei der Abarbeitung dynamisch durch den Interpretationsprozeß entstehenden plattformspezifischen Maschinencode ab und puffert ihn zwischen. Bei jeder erneuten Ausführung derselben Bytecodesequenz wird nun dieser bereits übersetzte Code ausgeführt. Dieses Vorgehen ist in den verbreiteten JVM-Implementierungen der Internet-Browser von Netscape und Microsoft verwirklicht. Ebenso bieten fast alle verfügbaren Java-Entwicklungsumgebungen diese Laufzeit-optimierende Komponente an. Der dadurch erzielte Geschwindigkeitsvorteil bewegt sich, je nach Struktur der Anwendung, zwischen zehn und 50 Prozent.

Den größten Geschwindigkeitszuwachs verspricht man sich jedoch von der vollständigen Realisierung der virtuellen Maschine in Hardware; damit wird sie de facto zur realen Maschine. Hierzu liegen jedoch noch keine Ergebnisse vor, welche die der derzeitigen Implementierung auf handelsüblichen Prozessoren signifikant übertrifft.

Eine andere Sichtweise nutzt das Bytecodeformat welches eine Zwischenrepräsentation darstellt nicht zur Interpretation mit dem Ziele der direkten Ausführung, sondern als Eingabeformat eines weiteren Übersetzungsschrittes, der üblicherweise plattformabhängigen nativen Code erzeugt. Für C++ existieren bereits Umsetzungen, die Bytecode in übersetzungsfähigen C++-Quellcode transformieren. Eine Spielart hiervon bildet das diskutierte Werkzeug javap dessen Ausgabeformat, nach einigen Umformungen, direkt als Eingabe weiterer Übersetzer akzeptiert wird.

**Web-Referenzen 4: Weiterführende Links**

- [Inside the Java Virtual Machine](#)

▲ **3 Betriebssysteme**

Mit der zunehmenden Verlagerung der Steuerungsaufgaben -- wie Bandwechsel, Auswahl des nächsten zu berechnenden Aufgabe sowie Überwachung des Maschinenstatus -- eines Rechners weg vom (menschlichen) Operateur hin zu dafür entwickelten Softwaresystemen entstand bereits in der Ära der [IBM 7094](#) der Begriff des *Betriebssystems* für Programme der genannten Funktionalität. (Vgl. [\[Cer03, S. 105\]](#))

3.1 Ablaufsteuerung

Eine der zentralen Aufgaben jedes Betriebssystems ist die Organisation und Verwaltung der aus einem Rechnersystem ausgeführten Programme. Die zuständige Betriebssystemkomponente wird daher zumeist mit *Ablaufsteuerung* bezeichnet.

Die Ablaufsteuerung organisiert die auszuführenden Aufgaben in Prozessen und organisiert deren Zuordnung zur CPU um Berechnungsergebnisse zu erhalten.

Prozesse und Threads

Aktuelle Rechnersysteme erlauben es verschiedene Aufgaben (scheinbar) zu einem Zeitpunkt auszuführen. So können gleichzeitig Dateien auf die Festplatte geschrieben werden und mit verschiedenen Applikationen (etwa Textverarbeitung, Web-Browser und MP3-Player) gearbeitet werden.

Voraussetzung dieses Verhaltens ist die Übertragung eines auf einem Speichermedium (Festplatte, CD-ROM, DVD oder Internet-Server) zur Verfügung stehenden Programms in den Hauptspeicher der Maschine und dessen Ausführung durch den (oder die) Prozessor(en).

Das Betriebssystem betrachtet alle auszuführenden Programme und Aufgaben einheitlich als *Prozeß* (oder *Task*), der über bestimmte festgelegte Eigenschaften verfügt und im selben Schema wie alle anderen Prozesse behandelt wird.

Gemäß der Anzahl der gleichzeitig verarbeitbaren Prozesse wird zwischen *Single Tasking*-Systemen, welche zu einem gegebenen Zeitpunkt nur genau einen einzigen Prozeß abarbeiten können, und *Multitasking* unterschieden. Letztere setzen durch schnelle Umschaltung zwischen den Einzelprozessen die sog. *Multiprogrammierung* um.

Solange das ausführende System jedoch nur eine gleichzeitig aktive Verarbeitungseinheit bietet, ist ausschließlich *nebenläufige Verarbeitung* (auch: *Pseudoparallelität*) realisierbar, welche durch schnelle Prozeßumschaltungen den Eindruck der simultanen Abarbeitung verschiedener Prozesse erweckt.

Reale Parallelität ist ausschließlich durch geeignete Hardwareunterstützung, d.h. durch die Bereitstellung duplizierter Funktionseinheiten erzielbar, die zu einem Zeitpunkt mehr als einen Prozeß ausführen.

Prozeßerzeugung

Zur Erzeugung eines neuen Prozesses müssen die auszuführenden Instruktionen (d.h. das Programm) sowie die zur Programmverarbeitung benötigten Hauptspeicherbereiche (Stack und Heap) verfügbar sein. Programmdateien und darauf operierender Code bilden zusammengekommen den *Prozeßkontext*.

Zusätzlich müssen betriebssystemseitig benötigte Verwaltungsdaten zur Kontrolle verschiedener Aspekte der laufenden Prozesse bereitgestellt werden. Dieser Datenbereich wird mit dem Sammelbegriff *Prozeßkontrollblock* oder auch *Prozeßtabelleneintrag* bezeichnet.

Jeder Eintrag in der Prozeßtabelle enthält diejenigen Daten, die benötigt werden um einen Prozeß verwalten können. Hierzu zählen insbesondere alle Angaben über den Zustand des Prozesses, die benötigt werden um ihn nach einem Anhalten (etwa durch Taskumschaltung) wieder lauffähig restaurieren zu können.

Daher umfaßt der Eintrag in der Prozeßtabelle alle prozessorspezifischen Daten, wie Registerbelegungen sowie den aktuellen Stand des Befehls- und Stackzeigers. Zusätzlich werden Daten über alle geöffneten Dateien sowie weitere betriebssystemspezifische Verwaltungsdaten (wie Benutzer- und Prozeß-ID) gespeichert.

Nach [\[Tan02, S. 95\]](#) umfaßt jeder Eintrag der Prozeßtabelle typischerweise die nachfolgend zusammengestellten Eigenschaften.

Prozeßverwaltung	Speicherverwaltung	Dateiverwaltung
Register	Zeiger auf Textsegment	Wurzelverzeichnis
Befehlszähler	Zeiger auf Datensegment	Arbeitsverzeichnis
Programmstatuswort	Zeiger auf Stacksegment	Dateideskriptor
Stackzeiger		Benutzer-ID
Prozeßzustand		Gruppen-ID
Priorität		
Scheduling-Parameter		
Prozeß-ID		
Elternprozeß		
Prozeßfamilie		
Signale		
Startzeit des Prozesses		
Verbrauchte CPU-Zeit		
CPU-Zeit der Kinder		
Zeitpunkt des nächsten Alarms		

Zur Erzeugung von Prozessen durch den Anwender stellen die Betriebssysteme entsprechende API-Funktionen zur

Verfügung, welche den auszuführenden Code in den Speicher laden und das Betriebssystem zur korrekten Initialisierung des Prozeßkontrollblockes veranlassen.

UNIX-artige Betriebssysteme wickeln die Prozeßerzeugung über die Funktion `fork` ab. Wird diese in einem Programm aufgerufen, so wird automatisch ein neuer Prozeß -- der *Kindprozeß* -- erzeugt. Dieser wird zunächst als Kopie des aktuell laufenden -- des *Elternprozeß* -- (d.h. desjenigen der `fork` aufruft) im Speicher angelegt, welche zwar über eigene lokale Variablen, sowie einen eigenen Stack verfügt, aber Zugriff auf dieselben Dateien besitzt wie der Elternprozeß.

Die Funktion `fork` regelt dabei über ihren Rückgabewert welche der beiden identischen Kopien als Eltern- und welche als Kindprozeß ausgeführt wird. So wird im Elternprozeß die Prozeßnummer des Kindprozesses zurückgeliefert, während die Funktionskopie des Kindprozesses als Rückgabewert 0 liefert.

Zunächst unterscheiden sich beide Prozeßinstanzen ausschließlich durch die Belegung des Rückgabewertes von `fork`.

Beispiel 10: Prozeßerzeugung unter UNIX



```
(1)int main (int argc, char** argv) {
(2)    printf("parent\n");
(3)    int i, pid;
(4)
(5)    if ((pid=fork()) == -1) {
(6)        printf("error creating process\n");
(7)        exit(1);
(8)    }
(9)
(10)   if (pid == 0) {
(11)       for (i=0; i<10; i++) {
(12)           sleep(1);
(13)           printf("child %d\n",i);
(14)       }
(15)   }
(16)   if (pid > 0) {
(17)       printf("child's pid=%d\n",pid);
(18)       for (i=0; i<10; i++) {
(19)           sleep(1);
(20)           printf("parent again %d\n",i);
(21)       }
(22)   }
(23)   return 0;
(24)}
```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Beispiel 10 zeigt die Anwendung der Systemfunktion `fork`.

Die lokalen Variablen `i` und `pid` stehen nach dem Aufruf von `fork` sowohl im Eltern- als auch im Kindprozeß zur Verfügung, jedoch verweisen sie auf unterschiedliche Speicherplätze um beiden Prozessen die unabhängige Belegung zu ermöglichen.

Insbesondere besitzt `pid` nach Aufruf und Duplikation der Codebereiche der beiden Prozesse im Elternprozeß den Wert der von 0 verschiedenen Prozeßnummer des Kindprozesses und setzt daher die Ausführung normal fort.

Im Kindprozeß hingegen ist `pid` mit 0 belegt und setzt daher mit der Abarbeitung in Zeile 11 fort und wird nach Abarbeitung der Schleife die ausschließlich im Elternprozeß ausgeführte Codesequenz der Zeilen 16 bis 22 überspringen.

Schlägt die Prozeßerzeugung fehl, so wird im erzeugenden Prozeß die Prozeßnummer -1 zurückgegeben; Ein Kindprozeß existiert dann naturgemäß nicht.

Solange der erzeugende Elternprozeß im System existiert, ist jeder Kindprozeß diesem eindeutig zugeordnet.

Hierdurch entsteht eine hierarchische Struktur aller Prozesse, die sich ausgehend vom Startprozeß `init` aufspannt.

Dieser Startprozeß nimmt eine Sonderrolle im System ein, da er als erster erzeugter Prozeß immer mit der feststehenden Prozeßnummer 1 versehen ist. Zusätzlich wird er durch die Abwesenheit eines Elternprozesses charakterisiert. Aus diesem Grunde ist `init` der nicht existente Elternprozeß mit der Prozeßnummer 0 zugeordnet.

Terminiert der Elternprozeß im Verlauf der Programmausführung vor dem Kindprozeß, so wird die Elternprozeßnummer im Kind auf die des `init`-Prozesses gesetzt. Kindprozesse, deren Elternprozeß vor ihnen terminiert, werden als *Waisen* (engl. *Orphan*) bezeichnet, die von `init` „adoptiert“ werden.

Diese Vorgehensweise wurde gewählt, da der erzeugende Elternprozeß über den Terminierungsstatus des Kindprozesses informiert wird. Dieser wird als `int`-Wert durch die im Kindprozeß aufgerufene Funktion `exit` übermittelt. Hierbei gilt unter UNIX die Besonderheit, daß die Binärrepräsentation des tatsächlich zurückgegebenen Wertes durch logisches Und mit 377 verknüpft wird.

Beendet der Kindprozeß seine Ausführung, ohne das der Elternprozeß den Rückgabewert übernimmt, so verbleibt der Kindprozeß als sog. *Zombie* bis zur Terminierung des Elternprozesses im Hauptspeicher.

Daher sollte ein erzeugender Prozeß unbedingt `wait` oder `waitpid` aufrufen um den Terminierungsstatus des Kindprozesses abzufragen und diesem somit die vollständige Terminierung zu ermöglichen.

Im Windows-System stehen die Standard-API-Funktionen `WaitForSingleObject` zum Warten auf das terminieren des erzeugten Kindprozesses bzw. `GetExitCodeProcess` zur Abfrage des Rückgabewertes zur Verfügung. Anders als unter UNIX retourniert Windows den numerischen Terminierungsstatus unmodifiziert an den erzeugenden Prozeß.

Oftmals ist es jedoch nicht gewünscht den vollständigen Code des Kindprozesses auch ungenutzt im Elternprozes zur

Verfügung zu stellen, und gleichzeitig umgekehrt im Kindprozess den dort ungenutzten Code des Elternprozesses, den dieser nach der Abspaltung mittels fork ausführt, vorzuhalten. Daher tritt ein Aufruf von fork zumeist in Gesellschaft mit einem Aufruf einer Funktion aus der exec-Familie auf.

Die Vertreter (typischerweise existieren mindestens fünf spezialisierte Varianten des exec-Aufrufs) dieser Funktionsgruppe ersetzen den gegenwärtig laufenden Prozeß vollständig durch einen anderen. Dessen Code muß zum Aufrufzeitpunkt noch nicht speicherresident sein, sondern kann durch die exec-Funktion nachgeladen werden. Beispiel 11 zeigt die Verwendung der Funktion [execlp](#) im Kindprozeß. Dessen Code wird durch den Funktionsaufruf in Zeile 14 vollständig durch den des Beispiels 12 überlagert.

Der Überlagerungsprozeß überschreibt den Prozeßkontext vollständig, so daß alle vom Elternprozeß duplizierten Daten und das aktuell ausgeführte Programm danach nicht mehr im Hauptspeicher zugreifbar sind. Aus diesem Grund wird typischerweise nicht der Aufruf fork zur Prozeßerzeugung, sondern [vfork](#) eingesetzt sofern direkt auf den fork-Vorgang ein exec-Aufruf folgt. Die vfork-Funktion besitzt den Vorteil, daß sie keine Duplikation der Datenbereiche des Elternprozesses vornimmt und daher deutlich schneller abgearbeitet werden kann als fork. In der Konsequenz darf der erzeugte Kindprozeß keine Daten verändern, da diese physisch mit dem Elternprozeß geteilt werden.

Beispiel 11: Prozeßerzeugung und Überlagerung mittels exec unter UNIX



```
(1)#include <stdlib.h>
(2)
(3)int main (int argc, char** argv) {
(4)    printf("parent\n");
(5)    int i, pid;
(6)    int execResult;
(7)
(8)    if ((pid=vfork()) == -1) {
(9)        printf("error creating process\n");
(10)       exit(1);
(11)    }
(12)
(13)    if (pid == 0) {
(14)        execResult = execlp("./child.exe", "child", NULL);
(15)        //only gets here in case of an error
(16)        printf("execlp returned: %d\n",execResult);
(17)    }
(18)    if (pid > 0) {
(19)        printf("child's pid=%d\n",pid);
(20)        for (i=0; i<10; i++) {
(21)            sleep(1);
(22)            printf("parent again %d\n",i);
(23)        }
(24)    }
(25)    return 0;
(26)}
```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Der unabhängig übersetzte Code, welcher durch den Kindprozeß nachgeladen wird ist in Beispiel 12 dargestellt:

Beispiel 12: Nachgeladener Code des Kindprozesses



```
(1)int main(int argc, char** argv) {
(2)    int i;
(3)
(4)    for (i=0; i<10; i++) {
(5)        printf("child %d\n",i);
(6)        sleep(1);
(7)    }
(8)}
```

[Download des Beispiels](#)

Zur Beschleunigung des fork-Aufrufes setzen Linux-Systeme die sog. *Copy-on-Write*-Technik ein. Hierbei werden die für den Kindprozeß bereitzustellenden Datenressourcen nicht direkt bei der Ausführung von fork dupliziert, sondern solange auf die Daten des Elternprozesses lesend zugegriffen, bis der Kind- oder Elternprozeß diese zu verändern wünscht. Erst dann (im Falle des schreibenden Zugriffs) müssen die Daten für den Kindprozeß dupliziert und separat bereitgestellt werden.

Hintergrund dieser Optimierungstechnik, die zu deutlichen Geschwindigkeitsgewinnen führen kann, ist die Erkenntnis, daß Kindprozesse nur in seltenen Fällen alle vom Elternprozeß übernommenen Daten verändern. In der überwiegenden Mehrzahl werden die duplizierten Daten hingegen ausschließlich lesend verarbeitet und nur ein kleiner Teil modifiziert.

Die Windows-Betriebssystemfamilie nimmt die Unterscheidung in Kindprozeßerzeugung und separatem Nachladen des benötigten Codes nicht vor, sondern bündelt beide Vorgänge in der Systemfunktion [CreateProcess](#).

Beispiel 13 zeigt den Aufruf der genannten Systemfunktion, welche den in Beispiel 14 dargestellten Code als Kindprozeß lädt.

Beispiel 13: Prozeßerzeugung unter Windows



```

(1)#include <Windows.h>
(2)
(3)int main( VOID ) {
(4)    STARTUPINFO si;
(5)    PROCESS_INFORMATION pi;
(6)    LPDWORD result;
(7)    int i;
(8)
(9)    ZeroMemory( &si, sizeof(si) );
(10)    si.cb = sizeof(si);
(11)    ZeroMemory( &pi, sizeof(pi) );
(12)
(13)    // Start the child process.
(14)    if( !CreateProcess( NULL, // No module name (use command line).
(15)        "childWin.exe", // Command line.
(16)        NULL,            // Process handle not inheritable.
(17)        NULL,            // Thread handle not inheritable.
(18)        FALSE,          // Set handle inheritance to FALSE.
(19)        0,              // No creation flags.
(20)        NULL,           // Use parent's environment block.
(21)        NULL,           // Use parent's starting directory.
(22)        &si,            // Pointer to STARTUPINFO structure.
(23)        &pi )          // Pointer to PROCESS_INFORMATION structure.
(24)    ) {
(25)        return 1;
(26)    }
(27)    for (i=0; i<10;i++) {
(28)        printf("parent %d\n",i);
(29)        Sleep(1000);
(30)    }
(31)    //wait for child to exit
(32)    WaitForSingleObject(pi.hProcess, INFINITE);
(33)    GetExitCodeProcess(pi.hProcess, result);
(34)    printf("child's exit code was: %i\n", *result);
(35)
(36)    return 0;
(37)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Beispiel 14: Quellcode des Kindprozesses



```

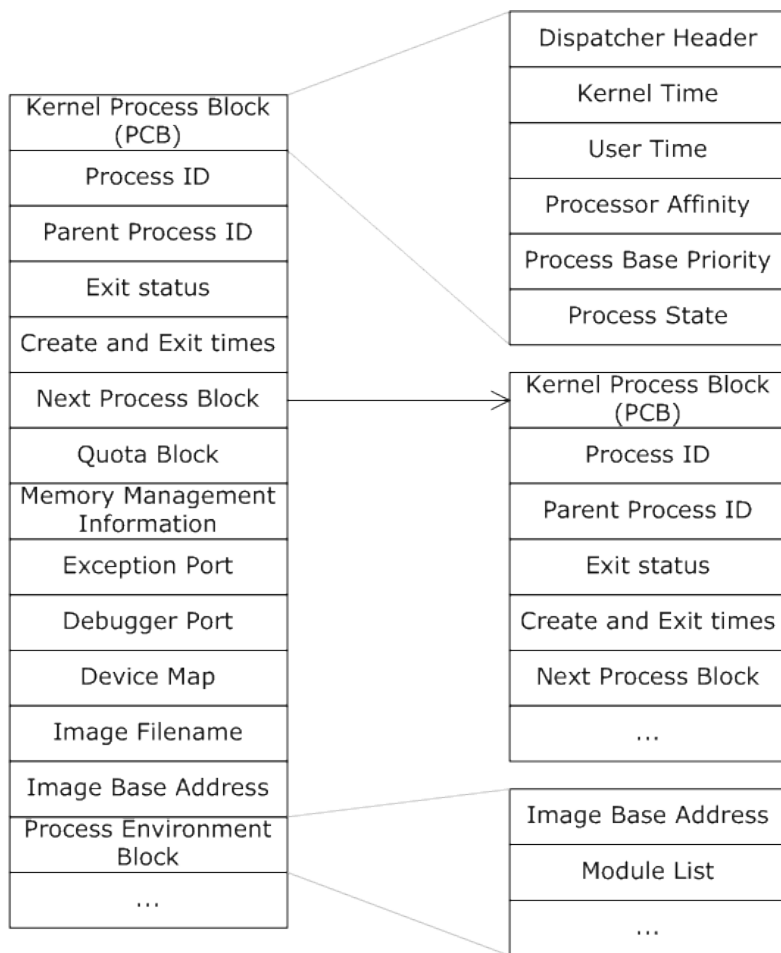
(1)#include <Windows.h>
(2)
(3)int main(int argc, char** argv) {
(4)    int i;
(5)    for (i=0; i<10; i++) {
(6)        printf("child %d\n",i);
(7)        Sleep(1000);
(8)    }
(9)    exit(2);
(10)}

```

[Download des Beispiels](#)

Die Struktur des Prozeßkontrollblocks (PCB), der auch als *Kernel Process Block* bezeichnet wird, ist in [Abbildung 16](#) dargestellt.

Abbildung 16: Aufbau des Windows-Prozeßkontrollblocks



(click on image to enlarge!)

Grundsätzlich enthält auch der unter Windows benutzte Prozeßkontrollblock die im vorhergehenden dargestellten Eigenschaften. Im Detail (beispielsweise an der Speicherung der geladenen DLL-Bibliotheken) lassen sich jedoch die Plattformbesonderheiten gut erkennen. (Vgl. hierzu auch: [\[Sol00, 279 u. 291\]](#))

Element	Beschreibung
Dispatcher Header	Verwaltungsinformation der Taskumschaltung
Kernel Time	Verbrauchte privilegierte Rechenzeit
User Time	Verbrauchte Anwenderrechenzeit
Processor Affinity	Zuweisung an genau eine CPU eines Multiprozessorsystems
Process Base Priority	Prozeßpriorität
Process State	Gegenwärtiger Prozeßzustand
Process ID	Eineindeutige Prozeßnummer
Parent Process ID	Prozeßnummer des Elternprozesses
Exit status	Abbruchgrund
Create and Exit times	Erzeugungs- und Terminierungszeitpunkt
Next Process Block	Verweis auf den nächsten Eintrag der Prozeßtabelle
Quota Block	Speichernutzungsbeschränkung
Memory Management Information	Verschiedene Aussagen über Nutzung des virtuellen Speichers
Exception Port	Kanal der Interprozeßkommunikation. Prozeßmanager sendet, bei Auftreten eines Ausnahmeereignisses, Nachricht dorthin.
Debugger Port	Kanal der Interprozeßkommunikation. Prozeßmanager sendet, bei Auftreten eines Debug-Ereignisses, Nachricht dorthin.
Device Map	Zuordnung der logischen Gerätenamen zur jeweiligen Hardware
Image Filename	Name der ausführbaren Binärdatei
Image Base Address	Startadresse der speicherresidenten ausführbaren Binärdatei
Module List	Verzeichnis der geladenen dynamischen Module (DLL-Dateien)

Die Tabelle zeigt im Kern wesentliche Gemeinsamkeiten mit der Auflistung der Prozeßkontrollfelder des UNIX-Systems

auf. So werden auch unter Windows alle im System verwalteten Prozesse einheitlich durch eine numerische Prozessnummer identifiziert. Ebenso besitzt jeder Prozeß (mit Ausnahme des Startprozesses `init` unter UNIX bzw. `Idle` unter Windows) einen eindeutigen Elternprozeß.

Zusätzlich zeigt die Tabelle einige systemspezifische Besonderheiten, wie beispielsweise die Verwaltung der geladenen DLL-Bibliotheksdateien.

Neben der direkten Möglichkeit Prozesse durch Betriebssystemfunktionen zu erzeugen, bieten hierfür auch Hochsprachenunterstützung an, wenngleich diese im engeren Sinne selbst keine Prozeßerzeugung vornehmen. Typischerweise bieten Hochsprachen lediglich einfach benutzbare Programmierschnittstellen als die nativen des Betriebssystems zur Prozeßerzeugung an. Die Aufrufe dieser Schnittstellen müssen jedoch zur Laufzeit auf die betriebssystemseitig verfügbaren Schnittstellen abgebildet werden. Dies kann bereits zur Übersetzungszeit durch den Compiler, oder zur Laufzeit durch den Interpreter, geschehen.

Im Falle der Java-Standard-API ist letzterer Ansatz verwirklicht, d.h. die virtuelle Java Maschine interpretiert zur Laufzeit den Hochsprachenauftrag und bildet ihn auf die entsprechende Betriebssystemfunktionalität ab. Innerhalb Javas stellt die Klasse [Runtime](#) verschiedene Varianten einer `exec`-Funktion zur Verfügung. Ausprägungen der Klasse `Runtime` fungieren hierbei zur Laufzeit als Schnittstelle zur betriebssystemseitigen Ausführungsumgebung, welche die Java Virtual Machine beherbergt. Daher ist zwar die direkt durch den Programmierer aufgerufene `exec`-Methode ebenso wie die durch sie aufgerufenen Methoden der Klasse [ProcessBuilder](#), welche die betriebssystemseitige Prozeßerzeugung kapselt, in Java realisiert, die Implementierung des `ProcessBuilders` durch die nicht als Bestandteil der Standard-API aufgefaßte Klasse `ProcessBuilderImpl` hingegen als plattformspezifische native Methode vorgenommen.

Daher verhält sich die durch `exec`-Javamethode so, wie die Kombination der UNIX-Systemaufrufe `fork` und `exec`, da die Javaplattform implizit die Erzeugung eines neuen Prozesses vornimmt und es damit durch `exec` zu keiner Überlagerung des ausgeführten Programmcodes kommt.

Ähnlich wie die gleichnamige UNIX-Systemfunktion gestatten Javas `exec`-Varianten ebenfalls den freien Aufruf externer ausführbarer Dateien, wie 15 zeigt.

Beispiel 15: Prozeßerzeugung unter Java



```
(1)import java.io.BufferedReader;
(2)import java.io.IOException;
(3)public class JavaExec {
(4)    public static void main(String args[]) {
(5)        Process p = null;
(6)        System.out.println("creating child");
(7)        Runtime rt = Runtime.getRuntime();
(8)        String command[] = new String[2];
(9)        command[0] = "java";
(10)       command[1] = "Child";
(11)       try {
(12)           p = rt.exec(command);
(13)       } catch (IOException ioe) {
(14)           System.out.println("IOException occurred");
(15)       }
(16)       try {
(17)           p.waitFor();
(18)       } catch (InterruptedException ie) {
(19)           System.out.println("InterruptedException occurred");
(20)       }
(21)       System.out.println("child has finished execution");
(22)       BufferedReader pOut = new BufferedReader(p.getInputStream());
(23)       byte output[] = null;
(24)       try {
(25)           output = new byte[pOut.available()];
(26)           pOut.read(output);
(27)       } catch (IOException ioe) {
(28)           System.out.println("IOException occurred");
(29)       }
(30)       System.out.println("output of the child process was: "
(31)           + new String(output));
(32)   }
(33)}
```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Der Code zeigt den Aufruf eines externen Programmes, am Beispiel der Erzeugung einer zweiten Java-Virtual-Machine-Instanz welche die Klasse [Child](#) zur Ausführung bringt.

Zur Erzeugung einer zusätzlichen Prozeßinstanz muß zunächst mittels der Methode [getRuntime](#) dasjenige Objekt ermittelt werden, welches die Verbindung zur betriebssystemseitigen Ausführungsumgebung repräsentiert.

Die auf Objekten dieses Typs ausführbare Methode `exec` gestattet dann die Übergabe einer Kommandozeile an die Betriebssystemumgebung. Im Beispiel wird die Zeichenkette `java Child` (aufgrund des separierten Leerzeichens in zwei `String`-Objekte aufgespalten) übergeben.

Der betriebssystemseitig erzeugte Prozeß wird unmittelbar nach seiner Erzeugung automatisch gestartet und abgearbeitet. Die Java-Methode liefert ein Objekt des Typs [Process](#) zurück, welches innerhalb der Hochsprache zur Kontrolle des erzeugten Prozesses eingesetzt werden kann.

Insbesondere kann auf die Terminierung durch Aufruf der Methode [waitFor](#) seitens des Erzeugers gewartet werden.

Als Besonderheit der Java-Plattform besitzen die erzeugten Kindprozesse keinen Zugriff auf die Standardkanäle für Standard-Ein- und -Ausgabe sowie Fehlerausgabe. Stattdessen werden alle Ausgaben des Kindprozesses an den erzeugenden Prozeß umgeleitet, von dem auch alle über die Standardeingabe vermittelten Eingaben erwartet werden. Aus diesem Grunde leitet der Elternprozeß die Ausgabe des Kindprozesses (sie ist Eingabe im erzeugenden Prozeß) mittels `getInputStream` um und gibt sie anschließend selbst aus.

Threads

Für viele Aufgaben, welche kurzfristig Nebenläufigkeitstechniken nutzen möchten, stellt der für die Prozeßerzeugung einzuplanende Zeitaufwand eine große Effizienzhürde dar. Insbesondere die Duplizierung des Elternprozeßkontextes nimmt, durch die notwendigen Kopieraufwände zur Bereitstellung der Variableninhalte auch für die Kindprozesse einen großen Zeitaufwand ein.

Darüber hinaus ist oftmals -- wie am Beispiel des Aufrufpaares `fork-exec` bzw. `CreateProcess` gezeigt -- ausschließlich die Bereitstellung zusätzlicher Ausführungsressourcen gewünscht, ohne die Datenressourcen gleichzeitig zu duplizieren.

Daher gilt grundsätzlich folgende Aufteilung (vgl. hierzu: [\[Tan02, S.98\]](#)):

Prozeßlokale Elemente	Threadlokale Elemente
Adressraum	Befehlszähler
Globale Variablen	Register
Geöffnete Dateien	Stack
Kindprozesse	Zustand
Signale	
Signalebehandlungsroutinen	
Abrechnungsdaten	

Aus diesen Gründen haben sich inzwischen separate Ausführungsfäden (sog. *Threads*) als leichtgewichtige Prozesse breit etabliert und den Begriff des *Multithreadings* in Erweiterung des klassischen Multiprogrammings geprägt. Dieser Prozeßtyp wird innerhalb genau eines Prozeßkontextes zur Ausführung gebracht. Daher kann jeder Thread auf die globalen Variablen des Elternprozesses zugreifen, besitzt jedoch einen eigenen Befehlszähler sowie Speicherplatz für den threadlokalen Stack und seine lokalen Variablen.

Durch die Beschränkung der zur Ausführung benötigten Ressourcen auf das absolute Minimum können Threads sehr schnell erzeugt werden und belasten daher die Gesamtleistungsfähigkeit des Systems nur marginal.

Im UNIX-Betriebssystem Linux steht für C und C++ die *POSIX Thread Library* (kurz: PThreads) zur Verfügung. Ihre durch den 1995 verabschiedeten Standard IEEE 1003.1c festgelegten Funktionen gestattet es dem Programmierer auf Hochsprachenebene eigenständig Threads zu erzeugen. Beispiel 16 zeigt die Umsetzung eines Thread-basierten C-Programmes.

Beispiel 16: Threads unter Linux mit PThreads

```
(1)#include <stdio.h>
(2)#include <pthread.h>
(3)
(4)int counter=0;
(5)
(6)void threadedFunction(void *ptr){
(7)    int i;
(8)    for (i=0; i<10; i++) {
(9)        printf("hello from %s (counter=%d)\n", (char*) ptr, counter++);
(10)        sleep(1);
(11)    }
(12)}
(13)
(14)int main(int argc, char** argv) {
(15)    pthread_t thread1, thread2;
(16)    int iret1, iret2;
(17)
(18)    iret1 = pthread_create( &thread1, NULL, (void*)&threadedFunction, (void*) "thread 1");
(19)    iret2 = pthread_create( &thread2, NULL, (void*)&threadedFunction, (void*) "thread 2");
(20)
(21)    pthread_join( thread1, NULL);
(22)    pthread_join( thread2, NULL);
(23)
(24)    printf("thread 1 returns: %d\n", iret1);
(25)    printf("thread 2 returns: %d\n", iret2);
(26)    exit(0);
(27)}
(28)
```



[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Die zur Umsetzung des Beispiels genutzte Bibliothek gestattet es, Nebenläufigkeit mit Funktionsgranularität zu realisieren. So wird im Beispiel die Funktion `threadedFunction` mittels des Aufrufes [pthread_create](#) als Programmfaden deklariert und ausgeführt.

Der Aufruf [pthread_join](#) sorgt dafür, daß mit der Weiterverarbeitung solange angehalten wird, bis der durch den übergebenen Zeiger identifizierte Thread seine Ausführung beendet hat.

Durch Nutzung der global deklarierten Variable `counter` zeigt der Beispielcode zusätzlich den gemeinsamen Zugriff auf Speicherbereiche durch die beiden erzeugten Threads.

Der Geschwindigkeitsvergleich zwischen zwei funktional äquivalenten Umsetzungen ([forkBench.c](#) und [pthreadBench.c](#)) zeigt deutlich, die Geschwindigkeitssteigerung beim Einsatz von Threads gegenüber schwergewichtigten herkömmlichen Prozessen.

So benötigt die Erzeugung und Terminierung von zehn Prozessen mittels `fork` 0.3 sec. während die selbe Anzahl Threads in 0.01 sec. angelegt und beendet werden kann. (Meßumgebung: Dual Athlon 1200, Linux 2.6.5)

Innerhalb der Windows-Betriebssystemumgebung wird zur Threaderzeugung die Standard-API-Funktion [CreateThread](#) angeboten. Sie operiert, wie der zuvor eingeführte POSIX-Aufruf, ebenfalls auf Funktionsebene. Der Code aus Beispiel 18 setzt die aus dem POSIX-Beispiel bekannte Funktionalität unter Nutzung der Windows-API um.

Beispiel 17: Threads unter Windows mit der Standard-API

```
(1)#include <windows.h>
(2)
(3)int counter=0;
(4)
(5)DWORD WINAPI ThreadFunc(LPVOID lpParam) {
(6)    int i;
(7)
(8)    for (i=0;i<10;i++)
(9)        printf("hello from thread %d (counter=%d)\n",*(DWORD*)lpParam, counter++);
(10)    return 0;
(11)}
(12)
(13)int main(VOID) {
(14)    DWORD dwThreadId1, dwThreadId2;
(15)    DWORD dwThrdParam1=1, dwThrdParam2=2;
(16)    HANDLE hThread1, hThread2;
(17)
(18)    hThread1 = CreateThread(
(19)        NULL,                                // default security attributes
(20)        0,                                    // use default stack size
(21)        ThreadFunc,                          // thread function
(22)        &dwThrdParam1,                      // argument to thread function
(23)        0,                                    // use default creation flags
(24)        &dwThreadId1);                      // returns the thread identifier
(25)
(26)    hThread2 = CreateThread(NULL,0,ThreadFunc,&dwThrdParam2,0,&dwThreadId2);
(27)
(28)    if (hThread1 == NULL || hThread2 == NULL) {
(29)        printf("CreateThread failed");
(30)    } else {
(31)        CloseHandle(hThread1);
(32)        CloseHandle(hThread2);
(33)    }
(34)    Sleep(100);
(35)    exit(0);
(36)}
```



[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Konventionsgemäß kann einer als Thread abzuarbeitenden Funktion in Windows lediglich genau ein Parameter des Typs `LPVOID` übergeben werden. Ein Verweis auf den erzeugten Thread wird innerhalb der Ausführung der Funktion `CreateThread` erzeugt und als Adresse an den Aufrufer zurückgeliefert.

Innerhalb Microsofts .NET-Umgebung vereinfacht sich der Umgang mit Betriebssystem-Threads deutlich. So kapselt, wie in Beispiel 1 dargestellt, die .NET-API durch die Klasse [ThreadPool](#) die Erzeugung eines Threads vollständig.

Die aus der Standard-API bekannte Einschränkung bei der Erzeugung nur höchstens einen Parameter übergeben zu können, ist jedoch auch in der .NET-Umsetzung unverändert präsent. Beispiel 1 setzt die aus den Beispielen 16 und 18 bekannte Funktionalität unter Verwendung des .NET-Frameworks um.

Beispiel 18: Threads unter Windows mit dem .NET-Framework



```
(1)using System;
(2)using System.Threading;
(3)
(4)class netThread {
(5)    private static int counter=0;
(6)    [STAThread]
(7)    static void Main(string[] args) {
(8)        ThreadPool.QueueUserWorkItem(new WaitCallback(myThread),"thread 1");
(9)        ThreadPool.QueueUserWorkItem(new WaitCallback(myThread),"thread 2");
(10)        Thread.Sleep(15000);
(11)    }
(12)
(13)    static void myThread(object threadName) {
(14)        for (int i=0;i<10;i++) {
(15)            Console.WriteLine("hello from "+threadName+ " (counter="+counter.ToString()
(16)            +")");
(17)            counter++;
(18)            Thread.Sleep(1000);
(19)        }
(20)    }
(21)}
```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Ähnlich zum herstellerseitig auf die Windows-Betriebssystemfamilie beschränkte .NET-Framework stellt auch die plattformunabhängig verfügbare Java-Umgebung eine Thread-Unterstützung auf Hochsprachenebene zur Verfügung. Ausgangspunkt der Threadunterstützung ist die Standard-API-Schnittstelle [Runnable](#).

Sie definiert die Signatur der Operation [run](#), welche als Startmethode eines Threads fungiert.

Anders als die Threadrealisierung durch die POSIX-Bibliothek und die in .NET umgesetzte Unterstützung gestattet Java ausschließlich die Bereitstellung von Nebenläufigkeit auf Klassenebene, da syntaktisch nur diese Schnittstellen implementieren können.

Beispiel 19 zeigt die Erzeugung und Ausführung von zwei separaten Threads, welche beide als Ausprägungen der Klasse `MyThread` realisiert sind.

Beispiel 19: Threederzeugung unter Java

```
(1)public class JavaThreadExample {
(2)    public static int counter = 0;
(3)    public static void main(String[] args) {
(4)        Thread t1 = new Thread(new MyThread("thread 1"));
(5)        Thread t2 = new Thread(new MyThread("thread 2"));
(6)        t1.start();
(7)        t2.start();
(8)        System.out.println("threads started ...");
(9)    }
(10)}
(11)class MyThread implements Runnable {
(12)    protected String text;
(13)    public MyThread(String text) {
(14)        this.text = text;
(15)    }
(16)    public void run() {
(17)        for (int i = 0; i < 10; i++) {
(18)            System.out.println("hello from " + text + " (counter="
(19)            + JavaThreadExample.counter + ")");
(20)            JavaThreadExample.counter++;
(21)            try {
(22)                Thread.sleep(1000);
(23)            } catch (InterruptedException ie) {
(24)                System.out.println("An InterruptedException occurred\n"
(25)                + ie.toString() + "\n" + ie.getMessage());
(26)            }
(27)        }
(28)    }
(29)}
```



[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Das Beispiel zeigt die Erzeugung zweier Threads, welche zunächst als gewöhnliche Java-Objekte vom Typ Klasse, welche `Runnable` implementiert, bereitgestellt werden.

Nach der Objekterzeugung muß die Ausführung je Thread explizit durch Aufruf der Methode [start](#) intiiert werden. Erst der Aufruf dieser Methode veranlaßt die virtuelle Java-Maschine dazu die im Thread-Objekt präsenste Methode `run` nebenläufig auszuführen.

Würde diese direkt aufgerufen werden, so erfolgt keine nebenläufige, sondern die herkömmliche synchrone, Ausführung.

Die Ausgabe des Beispiels zeigt gleichzeitig die Grenzen des Threadansatzes auf. Zwar verfügen alle Threads eines Prozeßkontextes über dieselben globalen Variablen, auf die diese unabhängig voneinander lesend und schreibend zugreifen können. Jedoch werden diese Zugriffe prinzipiell unsynchronisiert abgewickelt, wodurch es zu Datenverlust durch Überschreiben vorheriger Ergebnisse kommen kann. Abhilfe schaffen hier die im Abschnitt *Konkurrenz und Synchronisation* vorgestellten Synchronisationsmechanismen.

Insgesamt zeigt sich die durch Threads eingeführte Trennung von Ressourcen- und Rechenkapazitätsbündelung als sinnvoll. So kapseln aktuelle Betriebssysteme codeausführende Einheiten generell durch Threads und konzentrieren die Ressourcenverwaltung im umgebenden Prozeß. Daher besitzt sowohl unter Linux, als auch Windows, jeder Prozeß mindestens einen Thread.

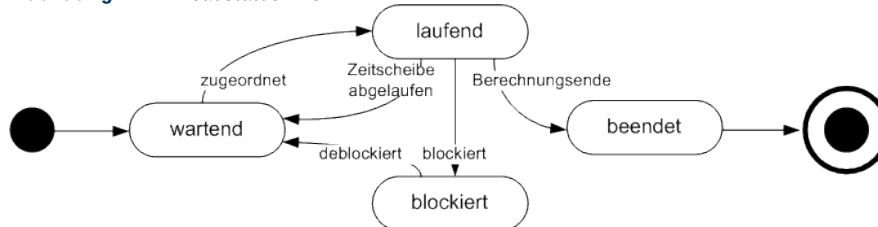
Prozessmodi

Zur Umsetzung der Parallelitätsillusion auf Rechnersystemen mit nur einer Berechnungseinheit ist die schnelle Umschaltung zwischen den rechenbereiten und -willigen Threads zwingend notwendig. Da sich je Berechnungseinheit jedoch zu einem Zeitpunkt nur genau ein Thread in Ausführung befinden kann, und die Anzahl der laufenden Rechengänge in einem System typischerweise größer ist als die Anzahl der zur Verfügung stehenden Hardware-Berechnungsressourcen, müssen einzelne Threads temporär von der Ausführung suspendiert werden, um anderen die Berechnung ihrer Ergebnisse zu ermöglichen. Die Selektion eines rechenbereiten Threads und Zuweisung zur Berechnungseinheit der Hardware obliegt dem sog. *Task Scheduler*.

Voraussetzung seiner Arbeit ist die Verwaltung von verschiedenen Threadstatus, die es ermöglichen zwischen rechnenden und von der Berechnung suspendierten Threads zu unterscheiden.

Im Betriebssystem UNIX werden generell die vier in [Abbildung 17](#) dargestellten Threadstatus unterschieden:

Abbildung 17: Threadstatus in UNIX



(click on image to enlarge!)

- **warten:** Ein Thread in diesem Status ist rechenbereit und ablauffähig. Die zur Verfügung stehenden Rechenheiten sind jedoch aktuell mit der Verarbeitung anderer Threads beschäftigt; Daher wird der Thread solange im Wartezustand gehalten, bis er einer Recheneinheit zugeteilt wird.
- **laufen:** Ein Thread rechnet aktuell, d.h. er ist genau einer Recheneinheit der Hardware zugeteilt und seine Instruktionen werden ausgeführt.
- **blockiert:** Die Ausführung eines Threads wurde unterbrochen weil der Thread auf ein externes Ereignis (z.B. Ein-/Ausgabe) wartet. Dieser Thread kann keiner Recheneinheit zugeteilt werden.
- **ende:** Der Thread hat seine Berechnungsaufgabe abgeschlossen und wird aus dem Hauptspeicher entfernt. Hierbei werden alle ihm zugeteilten Daten- und Speicherbereiche freigegeben.

Die Zustandsübergänge zwischen den einzelnen Status treten jeweils durch verschiedene Bedingungen und Ereignisse ein.

So befindet sich ein neu im System erzeugter Thread zunächst im Zustand *wartend*, da er nach seiner Erzeugung zunächst auf die erstmalige Zuteilung von Berechnungsressourcen durch das Betriebssystem warten muß. Wird ein Thread zur Verarbeitung selektiert, so wird er genau einer Berechnungseinheit *zugeordnet* und sein Code dort zur Ausführung gebracht. Hierdurch erfolgt der Zustandsübergang von *wartend* zu *laufend*. Ein Thread verbleibt solange in Ausführung, bis entweder seine zugeteilte Berechnungszeitscheibe abgelaufen ist oder er durch Warten auf ein externes Ereignis (z.B. Bereitstellung von Ein-/Ausgabedaten, Einlagern von Speicherseiten nach Seitenfehler) selbst die Berechnung einstellt. Im Falle des Entzugs der Berechnungsressourcen durch das Betriebssystem nach Ablauf der zugeteilten Rechenzeit geht der (prinzipiell unverändert rechenwillige und -bereite) Thread wieder in den Zustand *wartend* über und verbleibt dort bis zur erneuten Zuteilung einer Berechnungsressource. Suspendiert der Thread seine Ausführung aufgrund einer externen Wartebedingung vor Ablauf der zugeteilten Zeitscheibe so wird der Thread in den Zustand *blockiert* überführt. Dort verbleibt er, bis das externe Ereignis eingetreten ist und kann bis dorthin nicht wieder einer Berechnungsressource zugeordnet werden. Das bedeutet, er wird vorübergehend aus der Liste der rechenbereiten Threads (sie befinden sich alle im Zustand *wartend*) entfernt.

Nach dem Ende der Berechnung geht ein Thread in den Zustand *beendet* über. In diesem Zustand wird sein Ende an seinen zugeordneten Elternprozeß gemeldet und die mit dem Thread verbundenen Daten- und Programmbereiche aus dem Hauptspeicher entfernt. Abschließend werden die zugeordneten Verwaltungsdaten des Betriebssystems und -- sofern es sich um den letzten Thread eines Prozesses handelt -- auch der Prozeßkontrollblock geleert.

In [Abbildung 17](#) unberücksichtigt ist der mögliche Zustand *Zombie*, der eingenommen wird falls ein Thread zwar seine Ausführung beendet, aber sein Terminieren nicht an den Elternprozeß melden kann.

Üblicherweise wird in UNIX- und Windows-Betriebssystem der Zustandsübergang von *laufend* zu *wartend* automatisch zeitscheibengesteuert durch das Betriebssystem vollzogen (sog. *präemptives Multitasking*). In einigen Systemen (darunter die 16-Bit Windowsvarianten) ist der Prozeß selbst für die Rückgabe der Zeitscheibe und damit Kooperation mit anderen rechenwilligen Prozessen verantwortlich. Daher wird dieser Multitaskingtyp auch als *kooperatives Multitasking* bezeichnet.

Vermöge einer dafür vorgesehenen Betriebssystem-Funktion (bei Windows ist dies der Aufruf `yield`) muß ein Prozeß die Kontrolle explizit an das Betriebssystem übergeben um anderen Prozessen die Ausführung zu ermöglichen.

Durch den Aufruf `pause` kann ein UNIX-Thread sich selbst von der Ausführung suspendieren, bis er durch ein Signal

geweckt wird. Er wird daher nach dem Aufruf von `pause` nicht in den Zustand wartend überführt, sondern verbleibt in blockiert bis er durch ein Signal (=externes Ereignis) geweckt wird.

Der Code aus Beispiel 20 illustriert dies am Beispiel der Wartebedingung auf das Eintreffen des Signals `SIGUSR1`. Erst wenn dieses durch die Behandlungsroutine „catcher“ verarbeitet wurde, wird die Ausführung unmittelbar mit der auf die `pause`-Anweisung folgenden Instruktion fortgesetzt.

Beispiel 20: Manuelles Suspendieren der Ausführung

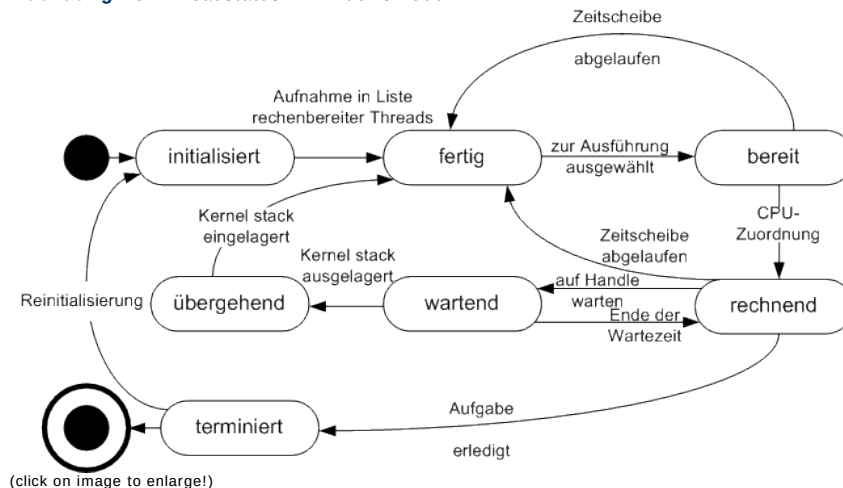
```
(1)#include <signal.h>
(2)
(3)void catcher() {
(4)    printf("SIGUSR1 caught\n");
(5)}
(6)
(7)int main(int argc, char** argv[]) {
(8)
(9)    signal(SIGUSR1, catcher);
(10)
(11)    printf("before\n");
(12)    pause();
(13)    printf("after\n");
(14)}
```



Download des Beispiels

Die Windows-Systemfamilie bietet in der Version *Windows 2000* ein dem UNIX-Threadzustandsmodell ähnliches Schema an. (Vgl. hierzu: [Sol00, S. 384f.]) Es unterscheidet zwischen sieben verschiedenen Status, die ein Thread einnehmen kann. Die zusätzlichen Zustände entstehen durch Aufspaltung UNIX-Zustände *wartend* und *blockiert*. Eine Übersicht der verschiedenen Zustände ist in [Abbildung 18](#) dargestellt.

Abbildung 18: Threadstatus in Windows 2000



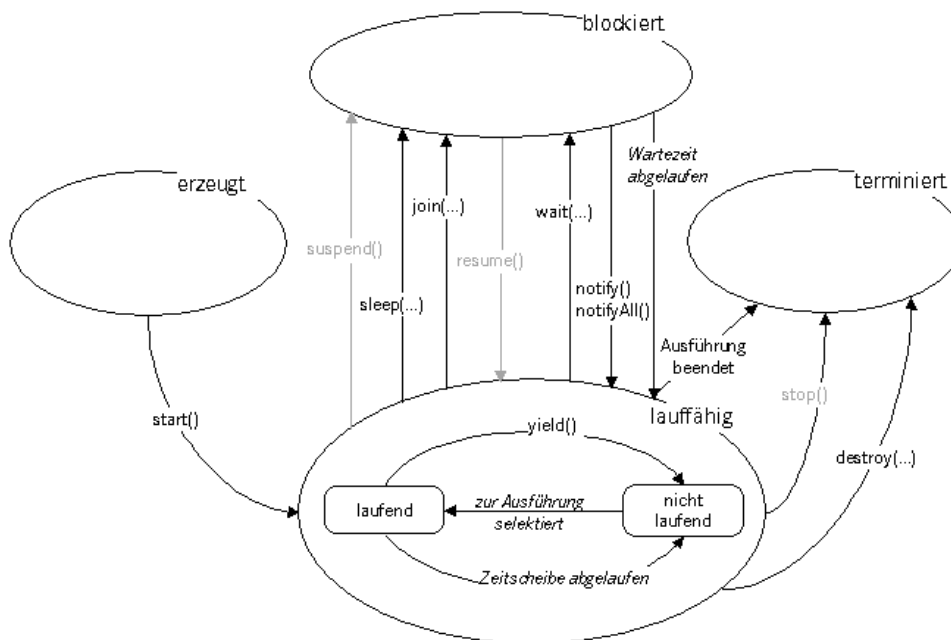
Die einzelnen Zustände sind hierbei:

- **initialisiert** (initialized): Betriebssysteminterner Übergangszustand während der Threaderzeugung.
- **fertig** (ready): Rechenbereite und vollständig speicherresidente Threads. Ausschließlich Threads in diesem Zustand können den Hardwarerecheneinheiten zugeteilt werden.
- **bereit** (standby): Ein in diesem Zustand befindlicher Thread wird der nächsten freiwerdenden CPU zugeordnet. Zu einem Zeitpunkt kann sich nur höchstens ein Thread in diesem Zustand befinden.
- **rechnend** (running): Ein Thread in diesem Zustand ist einer CPU zugeordnet und rechnet.
- **wartend** (waiting): Ein Thread kann durch Eintritt einer Wartebedingung in diesen Zustand versetzt werden. Eine solche Bedingung kann durch Warten auf das Ende einer Ein-/Ausgabeoperation oder durch freiwillige vorzeitige Rückgabe der Zeitscheibe gegeben sein.
- **übergehend** (transition): Ein Thread wird in diesen Zustand überführt, wenn er rechenfertig ist, jedoch threadspezifische Speicherbereiche des Kernels nicht Hauptspeicherresident sind.
- **terminiert** (terminated): Sind alle Instruktionen eines Threads ausgeführt, so geht er in diesen Zustand über. Ein Thread muß nach Verlassen dieses Zustands nicht zwingend aus dem Speicher entfernt werden, sondern kann auch neuinitialisiert und erneut ausgeführt werden.

Auch innerhalb der Virtuellen Java Maschine werden Threads in verschiedenen Ausführungsstatus verwaltet. Im Kern sind die hierbei unterschiedenen Phasen mit *erzeugt*, *lauffähig*, *blockiert* und *terminiert* stark an die bereits von UNIX bekannten Modi an. Jedoch stehen durch die verschiedenen Java-Methoden zur Threadverwaltung eine Reihe von Eingriffspunkten zur manuellen Beeinflussung der jeweiligen Zustandsübergänge durch den Programmierer zur Verfügung.

[Abbildung 19](#) zeigt die verschiedenen Zustandsübergänge sowie die Übergangsbedingungen.

Abbildung 19: Zustandsmodell der Java-Threads



(click on image to enlarge!)

Unterbrechungsverwaltung und Scheduling

Die Rechenbereitschaft und Zuordnung zu einer physischen Ausführungseinheit ist zwar notwendig, jedoch nicht hinreichend, um einem Thread dauerhaften CPU-Zugriff zu ermöglichen. Vielmehr existieren eine Reihe von Gründen, die -- obwohl die zu berechnende Aufgabe noch nicht erfüllt ist -- zum Entzug der CPU-Ressource führen können. In diesem Falle geht der Thread vom Zustand *laufend* in den Status *blockiert* über und kann zunächst nicht mehr zur Ausführung gelangen.

Zusammenfassend werden alle Ereignisse, welche zur Entziehung der CPU-Ressource eines Threads führen als *Unterbrechung* (engl. *Interrupt*) bezeichnet.

Zusätzlich werden Unterbrechungen hinsichtlich ihrer Ursachen in *synchrone Unterbrechungen* und *asynchrone Unterbrechungen* unterschieden.

Während synchrone Unterbrechungen innerhalb des ausgeführten Threads selbst durch den Programmcode verursacht werden liegt die Ursache asynchroner Unterbrechungen jenseits des Betriebssystems innerhalb der angeschlossenen Hardware.

Synchrone Unterbrechungen können sich in einem Instruktionsfluß durch die Struktur des ausgeführten Codes (z.B. Division durch Null) ergeben oder beabsichtigt, durch einen Systemaufruf, ausgelöst worden sein. Bei identischer Konfiguration, d.h. Ausführung desselben Programmes an der selben Speicheradresse unter Rekonstruktion des übrigen Systemzustandes, treten synchrone Unterbrechungen erneut in identischer Weise auf.

Asynchrone Unterbrechungen ergeben sich aus den dynamischen Zuständen der verwalteten Hardware und können durch Ein-/Ausgabe oder Auftreten des durch einen Hardwarebaustein periodisch generierten Uhrensymbols (Zeitgeber) erzeugt worden sein.

Die nachfolgende Tabelle gibt einen Überblick verschiedener Unterbrechungstypen und ihrer Ursachen.

Unterbrechung	Typ	Auslöser
Ein-/Ausgabeanforderung	asynchron	Geräte-Controller
Seitenfehler	synchron	Betriebssystem
Zeitgeber	asynchron	Uhrenbaustein
Aufruf von Systemfunktionen	synchron	Betriebssystem
Division durch Null	synchron	Betriebssystem
Arithmetik Über-/Unterlauf	synchron	Betriebssystem
Speicherschutzverletzung	synchron	Betriebssystem
Mausbewegung	asynchron	Maus
Tastendruck	asynchron	Tastatur
Netzwerkpaket	asynchron	Netzwerkkarte
USB	asynchron	USB-Controller
Stromausfall	asynchron	Stromversorgung

Synchrone Unterbrechungen wie *Seitenfehler*, *Aufruf von Systemfunktionen* und die Arithmetikfehler werden zwar durch die ausgeführten Programminstruktionen bedingt, Unterbrechungsauslöser ist jedoch das Betriebssystem selbst.

Die nähere Analyse der Tabelle zeigt, daß sich die darin aufgelisteten Unterbrechungen nicht nur hinsichtlich ihres Typs, sondern auch graduell im Hinblick auf die anzunehmende Dringlichkeit ihrer Behandlung unterscheiden lassen.

So werden Unterbrechungen zusätzlich zur getroffenen Einteilung in synchrone und asynchrone noch zusätzlich mit Prioritäten versehen. Beispielsweise kann die Behandlung eines Tastendrucks nicht die Verarbeitung eines Zeitgeberereignisses unterbrechen.

Zusätzlich zur Priorisierung von Unterbrechungen kann eine Unterbrechung, welche nicht durch andere Unterbrechungen in der Ausführung ihrer Behandlungsroutine unterbrochen werden darf, sich temporär als ununterbrechbar deklarieren um mit Teilen der Behandlung abzuschließen bevor sie durch weitere Unterbrechungen unterbrochen werden kann.

Im Kern stellt die Ununterbrechbarkeit lediglich eine besondere Form der Priorisierung dar, welche als so hoch einzustufen ist, daß keine anderen Unterbrechungen unterbrechend wirken können.

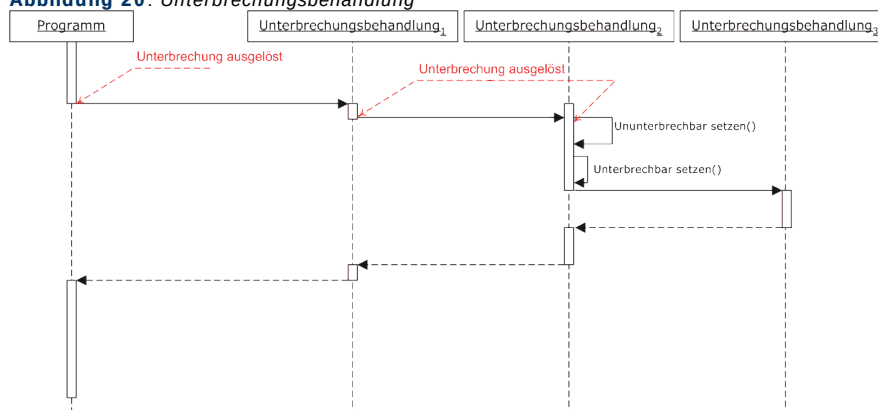
Im Falle des Auftretens einer Unterbrechung wird die aktuelle Programmausführung so lange suspendiert, bis die Unterbrechung behandelt wurde. Eine Ausnahme hiervon bilden lediglich niederpriorisierte Unterbrechungen, die versuchen höherpriorie zu suspendieren und ununterbrechbare Unterbrechungsbehandlungen.

[Abbildung 20](#) zeigt die Behandlung der Unterbrechungsbehandlung₁ welches einen Thread des in Ausführung befindlichen Programmes unterbricht. Während der Behandlung der Unterbrechung wird diese ihrerseits durch die Unterbrechungsbehandlung₂ unterbrochen und von der Ausführung suspendiert.

Unterbrechungsbehandlung₂ deklariert sich im Verlauf ihrer Ausführung als Ununterbrechbar. Daher wird das Auftreten der Unterbrechung₃ zunächst solange ignoriert, bis sich die laufende Unterbrechungsbehandlung₂ wieder als ununterbrechbar deklariert und durch die Unterbrechungsbehandlung₃ unterbrochen wird.

Nach Abschluß der Unterbrechungsbehandlung₃ kehrt diese Routine an diejenige Stelle zurück, in der Unterbrechungsbehandlung₂ verlassen wurde und schließt die Behandlung dieser Unterbrechung ab. Nach Ende der Unterbrechungsbehandlung₂ kehrt diese zur Unterbrechungsbehandlung₁ zurück, welche sie initial unterbrochen hat. Ist auch diese Unterbrechungsbehandlung abgeschlossen und wurde nicht zwischenzeitlich durch andere Unterbrechungen unterbrochen, so wird zur Programmausführung zurückgekehrt.

Abbildung 20: Unterbrechungsbehandlung



(click on image to enlarge!)

Scheduling

Zentrale Bedeutung in der Steuerung des Systemverhaltens kommt der Zeitgeberunterbrechung zu. In Betriebssystemen mit präemptivem Multitasking legt sie die technische Basis der Parallelitätssteuerung. Konkret wird jeder zur Ausführung selektierte Thread höchstens für eine feststehende Zeitspanne (die sog. *Zeitscheibe*) ausgeführt, sofern er nicht bereits vor Ablauf der maximalen Rechenzeit blockiert wurde, und nach Ablauf der Zeitscheibe wird ihm die CPU-Ressource entzogen und automatisch dem betriebssysteminternen *Scheduler*-Thread zugeteilt. Dieser während der gesamten Laufzeit des Betriebssystems aktive Thread entscheidet dann welcher Thread aus der Menge der rechenbereiten entnommen und als nächstes der CPU zugeteilt wird.

Die konkrete Auswahl eines Threads aus der Menge der um Ausführung konkurrierenden erfolgt nach festen Kriterien, die im *Schedulingalgorithmus* hinterlegt sind.

Grundsätzlich wird in aktuellen Betriebssystemimplementierungen ausschließlich die durch einen Prozeß verursachte CPU-Belastung als Kriterium der Prozessorzuteilung herangezogen und auftretende Ein-/Ausgabeoperationen vernachlässigt. Dieser Einschätzung liegt die Auffassung zu Grunde, daß das Geschwindigkeitsverhalten im wesentlichen durch die zu berechnenden Operationen determiniert wird. Die zunehmend leistungsfähigen Prozessoren offenbaren jedoch eine starke Abhängigkeit der Gesamtausführungsgeschwindigkeit von den durchzuführenden Lese- und Schreibzugriffen, da moderne Hauptprozessoren um einen Faktor 10.000 schneller Ergebnisse erzeugen, als sie Festplatten zu liefern vermögen. Daher wird auch das Ein-/Ausgabeverhalten zukünftig in die Zuteilungsstrategie der Threads eingang finden.

Nachfolgend werden die aktuell präsenten Basisansätze zur Steuerung der Parallelität auf Threadebene vorgestellt und abschließend an Beispielen aus realen Betriebssystemen diskutiert.

Die Kernfrage der Planung der Threadumschaltung bildet die Fragestellung wann eine Umschaltung zwischen zwei Threads erfolgen soll. Diese Entscheidung kann beim Eintreten verschiedener Ereignisse im System imminent werden:

- Threaderzeugung.
Nach der Erzeugung einer neuen rechenfähigen Verwaltungseinheit im Betriebssystem ist zu klären ob diese direkt ausgeführt wird, oder ob der erzeugende Thread zunächst zuende rechnet.
- Threadterminierung.
Beendet sich ein aktuell in Ausführung befindlicher Thread oder wird er durch ein externes Ereignis terminiert, so ist ein neuer rechenwilliger Thread zur Ausführung zu selektieren. Liegt ein solcher nicht vor, so wird dem Leerlaufthread (*idle*-Thread) die CPU zugeteilt.
- Threadblockade.
Wird ein Thread aufgrund einer Unterbrechung blockiert, so muß seine Ausführung suspendiert werden, bis die

Unterbrechung behandelt wurde.

- Ende einer Unterbrechungsbehandlung.
Ist die Behandlung einer Unterbrechung, die zur Blockade eines rechnenden Threads geführt hat, abgeschlossen, so ist der blockierte Thread wieder der Menge der rechenbereiten und -willigen zuzuordnen.
- Zeitgeberereignis.
Unterbrechung des aktuell laufenden Threads und Selektion eines rechenbereiten zur Ausführung.

Prinzipiell sollte sich die Auswahl eines geeigneten Schedulingalgorithmus an der Natur der Ausführungsumgebung orientieren. So muß der Threadumschaltung in batch-orientierten Umgebungen deutlich weniger Aufmerksamkeit zu Teil werden, als für hoch interaktive Systeme, da das Toleranzverhalten der wartenden Anwender bei Stapelverarbeitung deutlich ausgeprägter erscheint. Ebenso bedingen Echtzeitsysteme, welche in fixierten Zeitintervallen die Lieferung einer Antwort garantieren müssen deutlich stärkere Restriktionen an die Auswahl des nächsten rechenbereiten Prozesses als interaktive Workstation- oder Desktopsysteme.

Einen Überblick der spezifischen Zielsetzungen liefert die nachfolgende Zusammenstellung (vgl. [\[Tan02, S. 153\]](#)):

- Generell:
 - Fairness -- Jeder Thread erhält Rechenzeit
 - Offensichtlichkeit -- Vergabekriterien für Rechenzeit sind offengelegt und nachvollziehbar
 - Balance -- Alle Teile des Systems sind annähernd gleichmäßig ausgelastet
- Batch-Systeme:
 - Durchsatz -- Maximiere Auslastung der verfügbaren Ressourcen
 - Durchlaufzeit -- Minimiere Durchlaufzeit jedes Einzeljobs
 - CPU-Belastung -- Maximiere Auslastung der CPU
- Interaktive Systeme:
 - Antwortzeit -- Minimiere Antwortzeit für Benutzeranfragen
- **Echtzeitsysteme:**
 - Vorhersagbarkeit -- Garantie des Eintreffens einer Bedingung (z.B. Beendigung einer Berechnungsaufgabe) zu einem fixierten Zeitpunkt

In Batch-Systemen wird die Jobverarbeitung im ununterbrechbaren Modus angeboten, d.h. rechenwillige Jobs können bereits rechnende nicht präemptiv unterbrechen. Daher obliegt dem Scheduling Algorithmus „lediglich“ die Auswahl des nächsten auszuführenden Jobs aus der Warteschlange der rechenbereiten nach Beendigung der aktuell verarbeiteten Aufgabe.

First-Come First Served: Einfachster Algorithmus, der Jobs in der Reihenfolge ihres Eintritts in die Warteschlange zur Verarbeitung selektiert. Wird ein Job während seiner Ausführung blockiert, so wird er am Ende der Warteschlange eingereiht und muß auf die erneute Zuteilung der CPU warten bis alle vor ihm befindlichen Jobs zugeteilt worden sind. Dieser, grundsätzlich auf andere Systemtypen übertragbare, Zuteilungsalgorithmus läßt sich mit relativ wenig Aufwand implementieren und ausführen. Augenscheinlich erfüllt die Einordnung am Warteschlangende auch die gestellten Kriterien der Fairness und Offenheit. Allerdings birgt die Vorgehensweise auch einen gravierenden Nachteil in sich. So benachteiligt die First-Come First-Served-Vorgehensweise Ein-/Ausgabe-intensive Prozesse tendenziell, da diese nach jeder Blockierung am Ende der Warteschlange eingereiht werden. Die hierdurch eintretende Wartezeit verlängert sich zusätzlich proportional zur Systemlast, die direkt mit der Warteschlangenlänge korreliert ist. Beispiel 21 zeigt ein Beispiel für die Zuteilungsstrategie nach dem First-Come-First-Served-Prinzip.

Beispiel 21: Job-Scheduling nach dem First-Come-First-Served-Prinzip

Gegeben seien die Aufgaben j_1 , j_2 und j_3 mit den individuellen Laufzeiten

$l: l(j_1)=3, l(j_2)=1$ und $l(j_3)=2$.



Bei Zuteilung nach dem First-Come-First-Served-Prinzip ergeben sich daher als Wartezeiten $w(j_i)$:

$$w(j_1)=3$$

$$w(j_2)=w(j_1)+1=3+1=4$$

$$w(j_3)=w(j_2)+2=4+2=6$$

Daraus ergibt sich die mittlere Wartezeit pro Job als $(3+4+6)/3=4\frac{1}{3}$.

Shortest Job First: Diese Zuteilungsstrategie setzt voraus, daß die Laufzeit einer Aufgabe im Voraus bekannt ist und alle Jobs gleichzeitig vorliegen und sich die Menge der zu verarbeitenden Aufgaben während der Verarbeitung nicht ändert. Ist dies der Fall, so werden die Jobs in der aufsteigenden Reihenfolge ihrer Ausführungszeiten zur Ausführung selektiert. Diese Scheduling-Strategie liefert zuverlässig optimale Ergebnisse, ist jedoch aufgrund der Eingang aufgestellten Restriktionen nur schwer umzusetzen, da in der Praxis häufig auch während der Ausführung noch weitere Jobs hinzukommen, die bei der nächsten Zuteilungsrunde berücksichtigt werden müssen. Insbesondere kann die Ankunft eines „Kurzläufers“, d.h. Jobs mit kurzer Laufzeit, die Modifikation einer bereits erstellten Zuteilungsreihenfolge bedingen. Überdies ist die Laufzeit von Jobs vor ihrer Ausführung nur schwer zu bestimmen, da beispielsweise die Dauer von Ein-/Ausgabeoperationen nicht im allgemeinen Falle bestimmt werden kann.

Beispiel 22 zeigt ein Beispiel für die Zuteilungsstrategie nach dem Shortest-Job-First-Prinzip sowie die möglichen anderen Zuteilungsstrategien.

Beispiel 22: Job-Scheduling nach dem Shortest-Job-First-Prinzip

Gegeben seien die Aufgaben und Laufzeiten aus Beispiel 21.
Bei Zuteilung nach dem Shortest-Job-First-Prinzip ergibt sich daher die

Durchlaufreihenfolge $s_1 = \{j_2, j_3, j_1\}$.

Die Wartezeiten w für jeden Job bei dieser

Durchlaufreihenfolge ergeben sich als:

$$w(j_1) = w(j_3) + 3 = w(j_2) + 2 + 3 = 1 + 2 + 3 = 6$$

$$w(j_2) = 1$$

$$w(j_3) = w(j_2) + 2 = 1 + 2 = 3$$

Die mittlere Durchlaufzeit pro Job bei Anwendung der Abarbeitungsreihenfolge s_1 liegt daher bei $(1+2+3)/3 = 3^1/3$.

Bei der Alternativzuteilung nach dem First-Come-First-Served-Prinzip ergibt sich die in Beispiel 21 berechnete mittlere Durchlaufzeit als $(3+4+6)/3 = 4^1/3$.

Bei Wahl der Abarbeitungsreihenfolge $s_3 = \{j_1, j_3, j_2\}$

(dies entspricht der Zuteilungsstrategie *Longest-Job-First*) ergeben sich die individuellen Wartezeiten als $s_3 = \{j_1, j_3, j_2\}$:

$$w(j_1) = 3$$



$$w(j_2) = w(j_3) + 1 = w(j_1) + 2 + 1 = 3 + 2 + 1 = 6$$

$$w(j_3) = w(j_1) + 2 = 5$$

Die mittlere Durchlaufzeit pro Job bei Anwendung der Abarbeitungsreihenfolge s_3 liegt daher bei $(6+5+3)/3 = 4^2/3$.

Bei Wahl der Abarbeitungsreihenfolge j_2, j_1, j_3 ergeben sich die individuellen Wartezeiten als: $s_4 = \{j_2, j_1, j_3\}$:

$$w(j_1) = w(j_2) + 3 = 1 + 3 = 4$$

$$w(j_2) = 1$$

$$w(j_3) = w(j_1) + 2 = w(j_2) + 3 + 2 = 1 + 3 + 2 = 6$$

Die mittlere Durchlaufzeit pro Job bei Anwendung der Abarbeitungsreihenfolge s_4 liegt daher bei $(4+1+6)/3 = 3^1/3$.

Bei Wahl der Abarbeitungsreihenfolge j_3, j_1, j_2 ergeben sich die individuellen Wartezeiten als: $s_5 = \{j_3, j_1, j_2\}$:

$$w(j_1) = w(j_3) + 3 = 2 + 3 = 5$$

$$w(j_2) = w(j_1) + 1 = w(j_3) + 3 + 1 = 6$$

$$w(j_3) = 2$$

Die mittlere Durchlaufzeit pro Job bei Anwendung der Abarbeitungsreihenfolge s_5 liegt daher bei $(5+6+2)/3 = 4^1/3$.

Bei Wahl der Abarbeitungsreihenfolge j_3, j_2, j_1 ergeben sich die individuellen Wartezeiten als: $s_6 = \{j_3, j_2, j_1\}$:

$$w(j_1) = w(j_2) + 3 = w(j_3) + 1 + 3 = 2 + 1 + 3 = 6$$

$$w(j_2) = w(j_3) + 1 = 2 + 1 = 3$$

$$w(j_3) = 2$$

Die mittlere Durchlaufzeit pro Job bei Anwendung der Abarbeitungsreihenfolge s_6 liegt daher bei $(6+3+2)/3 = 3^1/3$.

Beispiel 22 zeigt, daß die Zuteilungsstrategie nach dem Shortest-Job-First-Prinzip immer ein optimales Ergebnis liefert, das die Wartezeit je Job verkürzt. Neben der durch die vorgestellte Strategie gefundenen Abarbeitungsreihenfolge können jedoch noch weitere existieren, welche dasselbe Optimum erreichen (im Beispiel sind dies s_4 und s_6), jedoch keine Alternativreihenfolge, welche das durch Bevorzugung des kürzestlaufenden Jobs ermittelte Optimum unterbieten würde.

Durch Modifikation des Shortest-Job-First-Prinzips zum **Shortest Remaining Job Next** läßt sich eine der beiden zentralen Einschränkungen des Shortest-Job-First-Algorithmus aufheben. Zwar erfordert auch die modifizierte Variante unverändert Kenntnis über die Laufzeit eines Jobs, jedoch wird der Zuteilungsvorgang nicht mehr nur genau einmal für eine Menge von Jobs ausgeführt, sondern die Zuteilung bei Ankunft eines neuen Jobs dynamisch neu vorgenommen.

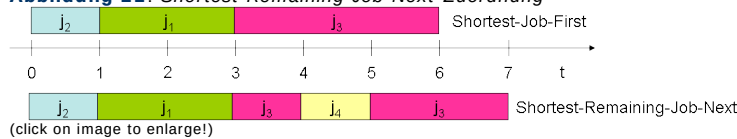
Der Zuteilungsalgorithmus unterscheidet sich daher nur unwesentlich vom Shortest-Job-First-Prinzip und legt den nächsten zu verarbeitenden Job bei Ankunft eines neuen neu fest. So können neuankommende Aufgaben bereits bestehende verdrängen, sofern sie eine Gesamtlaufzeit aufweisen, die kürzer als die Restlaufzeit des aktuell verarbeiteten Jobs ist.

Die Abarbeitungszeit des unterbrochenen Jobs verlängert sich dabei um die Dauer des „zwischengeschalteten“ Jobs, der ihn unterbrach.

Die Shortest-Remaining-Job-Strategie erweist sich als besonders Fair, was die Abarbeitung von (eigentlich bevorzugt zu verarbeitenden) kurzlaufenden Aufgaben betrifft und beseitigt deren -- durch verspätete Ankunft hervorgerufene -- Benachteiligung im Shortest-Job-First-Prinzip.

Abbildung 21 zeigt die Unterbrechung des aus den vorhergehenden Beispielen bekannten Jobs j_3 nach einer Zeiteinheit durch den neuankommenden Job j_4 der Dauer 1.

Abbildung 21: Shortest-Remaining-Job-Next-Zuordnung



Während alle bisher diskutierten Verfahren ausschließlich auf den Stapelverarbeitungsbetrieb zugeschnitten sind, läßt sich die *Shortest-Remaining-Job-Next*-Zuteilungsstrategie sowohl in Batch-orientierten als auch interaktiven Systemen einsetzen. Allerdings haben sich im Laufe der Zeit eine Reihe spezialisierter Zuteilungsalgorithmen für den Typus multitaskingfähiger interaktiver Systeme herausgebildet, die deren besondere Gegebenheiten besser berücksichtigen. Einige ausgewählte Basistechniken werden nachfolgend eingeführt.

Das älteste und daher in der Praxis auch am weitesten verbreitete Schedulingverfahren interaktiver Systeme ist unter dem Namen *Round Robin* geläufig.

Basisprinzip des Round-Robin-Verfahrens ist die Zuweisung eines fixierten Ausführungsquantums -- der Zeitscheibe -- an jeden Thread. Für die durch das Quantum festgelegte Zeitscheibe erhält jeder Thread exklusiv die CPU und wird nach Ablauf der zugewiesenen Zeitscheibe durch den Scheduler unterbrochen, sofern er nicht zuvor durch eine Unterbrechung blockiert wurde.

Nach Entzug der CPU wird der Thread in eine Warteschlange eingereiht und der nächst in der Schlange befindliche Thread erhält die CPU.

Round-Robin-basierte Prozessorzuteilung setzt keinerlei Kenntnisse über die Laufzeit eines Threads voraus und kann daher leicht implementiert werden. Die technische Umsetzung erfordert lediglich die Verwaltung einer Liste der Länge der maximal verwaltbaren Threadanzahl.

Wichtigster Freiheitsgrad des Round-Robin-Verfahrens ist die Länge des CPU-Quantums, welches einem Thread zugeteilt wird. Aufgrund des zu kalkulierenden Zeitaufwandes für die Threadumschaltung (Kontextwechsel) sollte die Zeitscheibe nicht zu gering festgelegt werden, um das Verhältnis zwischen Rechen- und Verwaltungszeit positiv zu halten. Gleichzeitig sollte die Zeitscheibe nicht zu großen Umfang annehmen, da dies die Interaktivität (d.h. die Wartezeit eines Threads auf CPU-Zuteilung) negativ beeinträchtigt.

Wird die Laufzeit l mit $l(t_1)=4$, $l(t_2)=1$ und $l(t_3)=2$ angenommen sowie eine Zeitscheibengröße von genau einer Zeiteinheit festgelegt, so ergibt sich der in Beispiel 23 dargestellte Ablauf.

Beispiel 23: Round-Robin-Scheduling

1. Rechnend: t_1

Warteschlange: $\{t_2, t_3\}$

2. Rechnend: t_2

Warteschlange: $\{t_3, t_1\}$

3. Rechnend: t_3

Warteschlange: $\{t_1\}$

Anmerkung: t_2 hat zu Ende gerechnet.

4. Rechnend: t_1



Warteschlange: $\{t_3\}$

5. Rechnend: t_3

Warteschlange: $\{t_1\}$

Anmerkung: t_3 hat zu Ende gerechnet.

6. Rechnend: t_1

Warteschlange: {}

7. Rechnend: t_1

Warteschlange: {}

Anmerkung: t_1 hat zu Ende gerechnet.

Das Beispiel illustriert die Gleichbehandlung aller im System aktiven Threads als weitere zentrale Eigenschaft des Round-Robin-Ansatzes. Dieses Verhalten ist oftmals nicht gewünscht. So sollten Threads, die wichtige Systemereignisse verarbeiten häufiger zugeordnet werden als gewöhnliche Anwenderthreads. Daher läßt sich das vorgestellte Verfahren durch die Zuordnung von Prioritäten zu jedem Thread dahingehend qualitativ verbessern, daß einzelne Threads bevorzugt zugeordnet werden.

Die Warteschlangenverarbeitung des Round-Robin-Algorithmus wird als Konsequenz von einer reinen Sortierung gemäß der Erzeugungsreihenfolge auf eine prioritätsbasierte Reihung umgestellt. Der Zuteilungsvorgang wählt unverändert das erste Element der Warteschlange aus und ordnet ihm die CPU-Ressource für den durch die Zeitscheibe vorgegebenen Zeitraum zu.

Um das „Verhungern“ niederpriorer Threads -- sie würden permanent hinter höherpriorien eingeordnet und erst nach deren Ende in der CPU-Verteilung berücksichtigt werden -- zu verhindern werden in der Praxis die Threadprioritäten dynamisch berechnet. Daher wird die *dynamische Priorität* des aktuell ausgeführten Prozesses nach dem Entzug der CPU, in Abhängigkeit seiner statischen Basispriorität, neu festgelegt.

Beispiel 24 zeigt die prioritätsbasierte Zuordnungsreihenfolge dreier Threads.

Beispiel 24: Prioritätsbasiertes Round-Robin-Scheduling

Gegeben seien drei mit Prioritätsstufen versehene Threads,

so daß gelte: $p(t_1)=3$, $p(t_2)=1$ und $p(t_3)=2$.

Die CPU-Zuteilung erfolge dynamisch prioritätsgesteuert.

Initial sei jeder Thread mit einer dynamischen Priorität von 10 versehen, von der die definierte Prioritätsstufe abgezogen wird. Dem Thread mit der numerisch höchsten Priorität wird anschließend die CPU zugeteilt. Nach Ablauf der Zeitscheibe oder Blockierung des Threads wird die dynamische Priorität um die definierte Prioritätsstufe vermindert.

Stehen im System mehrere Threads gleicher dynamischer Priorität zur Verfügung,

so wird bevorzugt einer ausgewählt, der noch nicht ausgeführt wurde. Steht kein Thread höherer dynamischer Priorität zur Verfügung, so wird der Thread, dem aktuell die CPU zugeteilt ist, weiter ausgeführt.

Nach einmaliger Zuteilung aller Threads beginnt das Verfahren von neuem.

1. Rechnend: t_2 (dynamische Priorität: $10-1=9$)



Wartend: t_1 (dynamische Priorität: $10-3=7$; noch nie ausgeführt), t_3 (dynamische Priorität: $10-2=8$; noch nie ausgeführt)

2. Rechnend: t_2 (dynamische Priorität: $9-1=8$)

Wartend: t_1 (dynamische Priorität: 7; noch nie ausgeführt), t_3 (dynamische Priorität: 8; noch nie ausgeführt)

Anmerkung: Verminderung der dynamischen Priorität des ausgeführten Threads, dynamische Priorität der wartenden Threads unverändert.

3. Rechnend: t_3 (dynamische Priorität: 8)

Wartend: t_1 (dynamische Priorität: 7), t_2 (dynamische Priorität: $8-1=7$)

Anmerkung: t_2 bei Kontextwechsel verdrängt, da t_3 höhere dynamische Priorität besitzt.

4. Rechnend: t_1 (dynamische Priorität: 7)

Wartend: t_2 (dynamische Priorität: 7), t_3 (dynamische Priorität: 6)

Anmerkung: t_1 verdrängt t_3 trotz „nur“ gleicher dynamischer Priorität, da t_1 noch nie ausgeführt wurde.

Nach dieser Zuteilung beginnt das Verfahren von neuem, da alle Threads einmal zugeordnet wurden.

Zur vereinfachten Handhabung bieten die Implementierungen des Round-Robin-Verfahrens typischerweise gestufte Prioritätsklassen an, in die einzelne Threads eingeordnet werden. Windows 2000 bietet beispielsweise 32 verschiedene Prioritätsstufen an, von denen 16 (Stufen 16-31) Echtzeitthreads, 15 (Stufen 1-15) Anwenderthreads und genau eine (Stufe 0) dem Leerlaufprozeß vorbehalten ist. Innerhalb der Anwenderprioritätsstufen wird zwischen *hochpriorien* (Klasse *high*), *höherpriorien* (Klasse *above normal*), *normalen* (Klasse *normal*), *niederpriorien* (Klasse *below normal*) und *niedrigpriorien* (Klasse *idle*) Threads unterschieden. Windows 2000 verwendet zur Abwicklung des Scheduling dynamische Prioritäten um eine faire Zuteilungsstrategie zu gewährleisten.

Beim Einsatz von Prioritätsklassen wird typischerweise prioritätsbasiertes Round-Robin-Scheduling zwischen den

Prioritätsklassen und Round-Robin-Verteilung innerhalb der Prioritätsklassen eingesetzt. Daher bedingt diese Umsetzungsform den Einsatz mehrerer eigenständiger Warteschlangen, die jeweils genau einer Prioritätsklasse zugeordnet sind.

Der zentrale Vorteil des Round-Robin-Verfahrens ist die einfache Umsetzung und die Unabhängigkeit von, typischerweise vorab nicht verfügbaren, Aussagen über die (erwartete) Laufzeit eines Threads. Allerdings böte die Berücksichtigung von Laufzeitaspekten eine weitere Quelle von Informationen über den Thread, die im Rahmen der Selektion des nächsten auszuführenden Threads berücksichtigt werden könnte. Im Grunde handelt es sich dabei um zusätzliche Information, die wie die Aussage über die bereits erfolgte CPU-Zuordnung im Schedulingalgorithmus der durch Beispiel 24 betrachtet wurde in die Auswertung einbezogen werden kann. Ebenso wie im Beispiel geschehen, könnten Aussagen über das zurückliegende Laufzeitverhalten eines Threads gesammelt und ausgewertet werden. Einen Ansatz hierzu bildet eine Übertragung des *Shortest-Job-First*-Verfahrens auf interaktive Systeme. Hierbei werden während der Ausführung Daten über CPU-Verweildauer gesammelt. Diese werden bei der dynamischen Festlegung der Priorität geeignet einbezogen, um Threads mit kurzer Laufzeit zu bevorzugen. Um ein realistisches Bild des veränderlichen Threadverhaltens zu erhalten bietet es sich an, nicht die gesamte bisherige Lebensdauer des Threads zur Beurteilung des Laufzeitverhaltens heranzuziehen, sondern lediglich auf jüngere Daten zurückzugreifen und diese geeignet zu gewichten. Einen Ansatz hierzu stellt die „Alterung“ (engl. *aging*) der gesammelten Daten dar, die ein gleitendes Fenster fester Größe zur Ermittlung der Einflußfaktoren auf die Prioritätsfestlegung heranziehen. Beispiel 25 zeigt die Berechnung dieses Einflußfaktors.

Beispiel 25: Aging

Gegeben seien drei mit Prioritäten versehene Threads, so daß gilt: $p(t_1)=2$, $p(t_2)=3$ und $p(t_3)=1$. Als Schedulingalgorithmus findet das in Beispiel 24 beschriebene Verfahren Anwendung. Zusätzlich wird aus den während der Laufzeit ermittelten CPU-Verweildauer eine zweite Priorität dynamisch ermittelt, welcher zur ersten addiert wird.

Thread	dynamische Priorität	Status
Schedulerlauf 1:		
t_1	$10-2+10-0=8+10=18$	wartend
t_2	$10-3+10-0=7+10=17$	wartend
t_3	$10-1+10-0=9+10=19$	rechnend
Schedulerlauf 2:		
t_1	18	wartend
t_2	17	rechnend
t_3	$9-1+10-0/2-7=8+3=11$	wartend
Schedulerlauf 3:		
t_1	18	rechnend
t_2	$7-3+10-0/2-2=4+8=12$	wartend
t_3	11	wartend
Schedulerlauf 4:		
t_1	$8-2+10-0/2+5=6+50=11$	wartend
t_2	12	rechnend
t_3	11	wartend
Schedulerlauf 5:		
t_1	11	wartend
t_2	$4-3+10-0/4+2/2-1=1+8=9$	wartend
t_3	11	rechnend

Während des ersten Schedulerlaufes liegen noch keine Daten über das dynamische Verhalten der (neu erzeugten) Threads vor, daher ist die zugehörige Prioritätskomponente durchgängig auf den Wert 0 gesetzt. Beim zweiten Durchlauf des Schedulers wird Thread t_2 zur Ausführung selektiert. Zusätzlich werden die gesammelten Ablaufdaten des ausgeführten Threads mit der Priorität 7 bewertet und bei der Neufestlegung der dynamischen Priorität entsprechend berücksichtigt.

Die dritte Anwendung des Schedulingalgorithmus liefert t_1 als Thread mit der höchsten dynamischen Priorität und teilt diesem die CPU zu.

Während seiner Ausführung werden auch über ihn dynamische Daten gesammelt, die beim vierten Schedulerlauf entsprechend in die Neufestlegung der dynamischen Priorität Eingang finden.

Der vierte Schedulerlauf wählt bereits zum zweiten Mal den mit der niedersten statischen Thread-Priorität versehenen t_3 zur Ausführung aus, da der Zuteilung-Algorithmus -- gestützt auf den gesammelten Verhaltensdaten -- seine erneute Selektion empfiehlt.

Der fünfte Durchlauf des Schedulers selektiert wahlfrei t_3 zur Ausführung, da seine dynamische Priorität identisch zu

t_3 ist und beide Thread bisher gleichhäufig ausgeführt wurden. Thread t_2 erhält die dynamische Priorität 9 zugeteilt. Gleichzeitig zeigt die Berechnung seiner Priorität die Wirkung des *agings*, bei dem mehrere historische Werte, nach ihrem Alter gewichtet, in die Berechnung einbezogen werden.

Insgesamt zeigt das Beispiel auch die Wirkungsmacht des Einbezugs von Laufzeitinformation in die Prioritätsfestlegung. So wird Thread t_2 -- obwohl er über die niedrigste statische Priorität verfügt --, vermöge der positiven Laufzeitdaten, ebenso häufig ausgeführt wie t_3 , der mit der höchsten statischen Priorität ausgestattet wurde.

Eine alternative Schedulingvariante ergibt sich, wenn die Zielsetzung der größtmöglichen Fairness in den Vordergrund gerückt wird. So ergibt sich durch die Verfolgung der Zielsetzung jedem Thread möglichst denselben Anteil an CPU-Zeit zuzuteilen eine **Scheduling-Garantie** auf die sich jeder Anwender verlassen kann.

Als Ziel kann hierbei verfolgt werden jedem Thread relativ zu seiner Lebenszeit dieselbe Menge Zuordnungen zuteil werden zu lassen. Beispiel 26 zeigt einen exemplarischen Ablauf:

Beispiel 26:

Thread	Alter	bisherige Zuteilungen	Status
Schedulerauf 1:			
t_1	1	0	rechnend
Schedulerauf 2:			
t_1	2	1	rechnend
Schedulerauf 3:			
t_1	3	2	wartend
t_2	1	0	rechnend
Schedulerauf 4:			
t_1	4	2	rechnend
t_2	2	1	wartend
Schedulerauf 5:			
t_1	5	3	wartend
t_2	3	1	rechnend
Schedulerauf 6:			
t_1	4	2	rechnend
t_2	3	2	wartend

Das Beispiel zeigt fünf aufeinanderfolgende Zuordnungen unter Anwendung garantierter Zuteilung. Zur CPU-Zuordnung ausgewählt wird jeweils der Thread, dessen Alter (d.h. Verweildauer im System) gegenüber den bisher erfolgten Zuteilungen am kleinsten ist.

Die dargestellte Schedulingvariante berücksichtigt ausschließlich die Anzahl der laufenden Threads und verspricht diesen Gleichbehandlung. Eine unter dem Namen **Fair-Share Scheduling** bekannte Variante dieses Vorgehens betrachtet nicht die Anzahl der Threads, sondern die Anzahl der sie besitzenden Anwender und garantiert die Gleichbehandlung aller Anwender statt der durch sie gestarteten Berechnungsaufgaben.

Ziel dieser Modifikation des Algorithmus ist es die inhärente Bevorzugung derjenigen Anwender, die viele Threads starten, zu eliminieren.

Eine Umsetzungsmöglichkeit wäre daher die Nutzung von garantiertem Scheduling zwischen den Anwendern und Zuordnung der Threads eines Anwenders im Round-Robin-Verfahren.

Das **Lotterie-Scheduling** greift die Idee der Fairness auf und steigert das Versprechen des Algorithmus der garantierten Zuordnung sogar noch. So werden beim Lotterie-ähnlichen Scheduling Lose an alle im System laufenden Threads verteilt. Jede zu vergebende Zeitscheibe wird unter den rechenwilligen Threads „verlost“. Gewinnt das Los eines Threads, so erhält er die CPU zugeordnet.

Der zugrundeliegende Algorithmus ist vergleichsweise einfach umzusetzen und gestattet sogar, durch die Verteilung von mehr als einem Los an einen Thread, die Nachbildung von Prioritäten. Allerdings hängt die praktische Einsetzbarkeit stark von der Qualität der zur Verfügung stehenden Zufallszahlen, welche die Basis des „Glücksspiels“ bilden ab. Sind diese nicht gleichverteilt, sondern treten gehäuft oder gar periodisch auf, so kann dies zur unerwünschten Bevorzugung einzelner Threads führen.

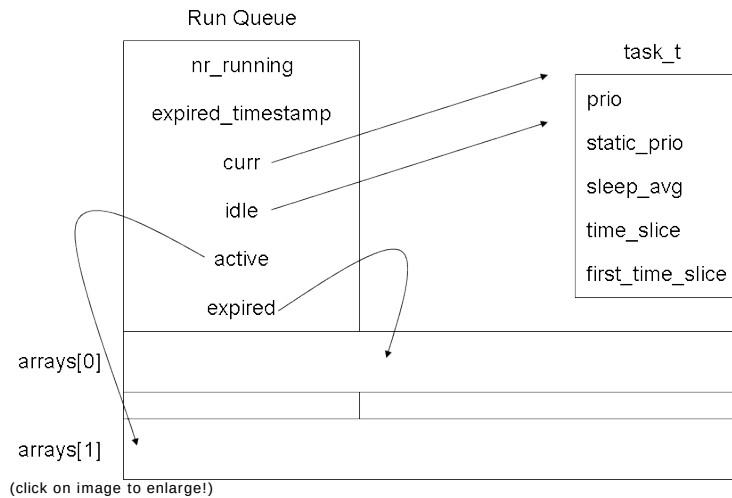
Die Windows 2000 Systemfamilie setzt eine Variante des garantierten Scheduling ein, welche jedem im System laufenden Thread grundsätzlich denselben CPU-Anteil (Quantum) garantiert. Zusätzlich verfügt jeder Thread über eine Prioritätsebene, welche seine Zuordnungshäufigkeit steuert. Die Workstations-Editionen der Windows-Betriebssysteme sehen zusätzlich, beispielsweise zur Steigerung derjenigen Threads die einem Programm zugeordnet sind, mit welchem der Anwender aktuell arbeitet, Prioritäts-Boosts vor, die kurzfristig die Priorität eines Threads steigern können.

Das Linux-Betriebssystem verfügt seit der Version 2.6 über einen neuen Scheduleransatz, welcher die faire CPU-Zuteilung auch bei hohen Systemlasten garantieren soll. Da Linux im Betriebssystemkern ausschließlich Prozesse behandelt berücksichtigt der Schedulingalgorithmus auch ausschließlich diese als kleine Einheit der Rechenzeitverteilung.

Ziel des Schedulingalgorithmus ist es größtmögliche Skalierbarkeit zu verwirklichen und die CPU-Zuteilung in konstanten Zeitintervallen zu garantieren. Der Schedulingalgorithmus besitzt daher die Komplexität $O(1)$.

Grundlegende Datenstruktur des Schedulers ist die in [Abbildung 22](#) zusammengestellt.

Abbildung 22: Datenstruktur des Linux-Schedulers



TODO:

Hier geht's demnächst weiter ...



Web-Referenzen 5: Weiterführende Links

- [PThreads-Tutorial](#)
- [YoLinux Tutorial: POSIX thread \(pthread\) libraries](#)
- [Vorlesung Java Threads](#)

▲ Schlagwortverzeichnis

[Ablaufsteuerung](#)

[ALU](#)

[arithmetic and logic unit](#)

[Array-Prozessoren](#)

[asynchrone Unterbrechungen](#)

[batch-Betrieb](#)

[buffer overflow](#)

[CISC](#)

[Complex Instruction Set Computing](#)

[control unit](#)

[Copy-on-Write](#)

[CP/M](#)

[CPU](#)

[dynamische Priorität](#)

[Elternprozeß](#)

[Emulation](#)

[enge Kopplung](#)

[EPIC](#)

[Explicit Parallel Instruction Computing](#)

[Fair-Share Scheduling](#)

[Feldrechner](#)

[Ferritkernspeicher](#)

[First-Come First Served](#)

[GUI](#)

[Halbaddierer](#)

[Hollerithmaschine](#)

[Instruction Pointer Register](#)

[Interrupt](#)

[jobs](#)

[Kindprozeß](#)

[Kontextwechsel](#)

[kooperatives Multitasking](#)

[leichtgewichtige Prozesse](#)

[Leitwerk](#)

[Linux](#)
[Iose Kopplung](#)
[Mainframe](#)
[Maus](#)
[Mikrocomputer](#)
[Mikrooperationen](#)
[Minicomputer](#)
[Moore'schen Gesetz](#)
[MS-DOS](#)
[MULTIC](#)
[Multiprogrammierung](#)
[Multiprogramming](#)
[Multitasking](#)
[Multithreading](#)
[nebenläufig](#)
[Non-Unified Memory Access](#)
[NUMA](#)
[Orphan](#)
[Parallelität](#)
[PC](#)
[Personal Computer](#)
[Phasen-Pipeline-Architektur](#)
[pipeline bubble](#)
[pipeline hazard](#)
[präemptives Multitasking](#)
[Prozeß](#)
[Prozeßkontext](#)
[Prozeßkontrollblock](#)
[Prozeßtabelle](#)
[Pseudoparallelität](#)
[Rechenwerk](#)
[Reduced Instruction Set Computing](#)
[Ringkernspeicher](#)
[Ripple-Carry-Addierer](#)
[RISC](#)
[Round Robin](#)
[Scheduler](#)
[Schedulingalgorithmus](#)
[Shortest Job First](#)
[Shortest Remaining Job Next](#)
[SIMD-Rechner](#)
[Single Tasking](#)
[SISD-Rechner](#)
[speedup](#)
[Spooling](#)
[Stapelverarbeitung](#)
[Stapelverarbeitungsbetrieb](#)
[Symmetrisches Multiprocessing](#)
[synchrone Unterbrechung](#)
[Task](#)
[Task Scheduler](#)
[Thread](#)
[Timesharing](#)
[UMA](#)
[Unified Memory Access](#)
[UNIX](#)
[Unterbrechung](#)
[Vektorprozessoren](#)
[Very Large Instruction Word](#)
[VLIW](#)
[von-Neumann-Architektur](#)
[Waisen](#)
[Windows](#)
[Zeitscheibe](#)
[Zentralprozessor](#)
[Zombie](#)

Service provided by [Mario Jeckle](#)

Generated: 2004-06-07T10:01:37+01:00

 [Feedback](#)  [SiteMap](#)

 [This page's original location: http://www.jeckle.de/vorlesung/sysarch/script.html](#)

 [RDF description for this page](#)

▲ Vorlesungen

▶ Scriptum

▶ Aktuelles **XML**

▶ Evaluierung

▼ Abstract

▼ Lernziel

▼ Notenbildung

▼ Vorlesungsinhalt

▼ Download des Scriptums

▼ Links

▼ Empfohlene Literatur

▼ Mailingliste zur Vorlesung

▲ Organisatorisches

Die Vorlesung findet donnerstags von 7.45 bis 11 Uhr im Raum A314 statt.

Die schriftliche Abschlußklausur findet am 2004-07-05 um 9.30 Uhr im Raum A314 statt.

An folgenden Terminen entfällt die Vorlesung:

- 2004-04-22
- 2004-04-29
- 2004-05-13
- 2004-05-20 (Christi Himmelfahrt)
- 2004-05-27
- 2004-06-03 (vorlesungsfreie Zeit nach Pfingsten)
- 2004-06-10 (Frohnleichnam)
- 2004-06-24

Dafür findet an folgenden Ersatzterminen Vorlesung jeweils ab 7.45 Uhr im Raum A232 statt:

- 2004-04-19
- 2004-04-26
- 2004-05-10
- 2004-05-17
- 2004-06-21
- 2004-06-28

Abstract

Streitfragen um die „richtige“ Architektur von Betriebssystemen und Rechnern bestimmen vielfach die (ver-)öffentliche Meinungsbildung und die Werbeaussagen großer Hersteller. Jenseits der plakativen und vermeindlichen Vergleiche und Entwicklungssprünge haben sich jedoch die der Rechnerarchitektur und dem Aufbau von Betriebssystem zugrundeliegenden Konzepte in den letzten Jahren nicht umwälzend geändert. Vielmehr existiert eine stabile Basis gut verstandener und in der Praxis erprobter Grundlagen, die auch in modernen Rechnerarchitekturen und Betriebssystemen Anwendung finden. Hierzu zählt der Bestand an Wissen über Aufbau und interne Organisation von Prozessoren und Betriebssystemen, hierbei insbesondere Aspekte der Ablaufsteuerung, Speicherverwaltung und des Zugriffs auf Hintergrundspeicher. Gleichzeitig eroberten sich verteilte Systeme in der jüngeren Vergangenheit einen solch zentralen Stellenwert in der Anwendung, daß Kenntnisse über deren Basismechanismen inzwischen auch als Grundlagenwissen anzusehen sind.

Lernziel

Der Student soll nach der Veranstaltung ...

- die Aufgaben der zentralen Komponenten einer Rechnerarchitektur kennen
- die Aufgaben eines Betriebssystems kennen
- verschiedene Realisierungsalternativen und existierende Systeme vergleichen und bewerten können

Notenbildung

Die Notenbildung erfolgt anteilig durch die im theoretischen und praktischen Vorlesungsteil erzielte Note im Verhältnis 2:1.

Inhaltsübersicht

- 1 Motivation und Einführung
 - 1.1 Rechner- und Betriebssystementwicklung
 - 1.2 Peripherie
 - 1.3 Zentralprozessoren
- 2 Rechnerarchitekturen
 - 2.1 Befehlsarchitekturen
 - 2.1.1 CISC-Befehlssatzarchitekturen
 - 2.1.2 RISC-Technik
 - 2.1.3 Expliziter Parallelismus und EPIC/VLIW
 - 2.2 Mikroarchitekturen

- 2.2.1 Phasen-Pipelining
- 2.2.2 Superskalarität
- 2.2.3 Multithreading-CPU's
- 2.3 Parallelverarbeitung
 - 2.3.1 Rechnerklassifikation
 - 2.3.2 Speichergekoppelte-Umgebungen
 - 2.3.3 Nachrichtengekoppelte-Umgebungen
- 2.4 Fallstudie: Die virtuelle Rechnerarchitektur der Java-Maschine
- 3 Betriebssysteme
 - 3.1 Ablaufsteuerung
 - 3.1.1 Prozesse und Threads
 - 3.1.2 Prozessmodi
 - 3.1.3 Unterbrechungsverwaltung und Scheduling
 - 3.2 Speicherverwaltung
 - 3.2.1 Speichermanagement
 - 3.2.2 Virtuelle Speichertechnik
 - 3.3 Konkurrenz und Synchronisation
 - 3.3.1 Dead- und Livelocks
 - 3.3.2 Synchronisationskonzepte
 - 3.4 Ein-/Ausgabesystem
 - 3.4.1 Physikalische E/A-System
 - 3.4.2 Logisches E/A-System
- 4 Verteilungsaspekte
 - 4.1 Kommunikation
 - 4.1.1 Nachrichtenvermittlung
 - 4.1.2 Entfernte Unterprogrammaufrufe (RPC)
 - 4.1.3 Client-Server-Modell
 - 4.2 Synchronisationsaspekte
 - 4.2.1 Uhrensynchronisation
 - 4.2.2 Wechselseitiger Ausschluß
 - 4.2.3 Transaktionen
 - 4.3 Verteilter Datenzugriff
 - 4.3.1 Dateisysteme
 - 4.3.2 Service-Orientierung
 - 4.3.3 Data Grids

Download des Scriptums

- [PDF](#)
- [PDF \(tief verlinkt\)](#)
- [Scriptquelle im XML-Format](#)
- [Code-Beispiele des Scriptums](#)

Links

Allgemeines

- [A History of Apple's Operating Systems](#)

- [Linux Tutorial](#)

Linux

- [Des Kaisers neue Kleider](#)
- [The post-halloween document](#)

▲ Empfohlene Literatur

Bücher

Hinweis: Diese Liste ist noch unvollständig!

- [Scriptum zur Vorlesung](#)
- M. J. Bach: *UNIX -- Wie funktioniert das Betriebssystem?*, Hanser, 1991.
- P. E. Ceruzzi: *Eine kleine Geschichte der EDV*, MITP, 2003.
- [B. Goodheart, J. Cox](#): *UNIX, System V, Release 4 -- Reise durch den Zaubergarten*, Prentice Hall, 1994.
- J. A. Harris: [Betriebssysteme -- 330 praxisnahe Übungen mit Lösungen](#), MITP, 2003.
- C. Mörtin: *Rechnerarchitekturen*, Fachbuchverlag Leipzig, 2001.
- [W. Mauerer](#): *Linux Kernelarchitektur*, Hanser Verlag, 2004.
- D. A. Solomon, M. E. Russinovich: *Inside Microsoft Windows 2000*, 3rd ed., Microsoft Press, 2000.
- A. S. Tanenbaum: *Moderne Betriebssysteme*, Pearson Studium, 2002.
Deutsche Übersetzung von: [Modern Operating Systems, Prentice Hall, 2001.](#)

Rechnerarchitektur

Intel IA-32-Architektur:

- [Optimization Reference Manual](#)
- Software Developer's Manual
 - [Volume 1: Basic Architecture](#)
 - [Volume 2A: Instruction Set Reference, A-M](#)
 - [Volume 2B: Instruction Set Reference, N-Z](#)
 - [Volume 3: System Programming Guide](#)

Intel IA-64-Architektur:

- [Itanium 2 Hardware Developer's Manual](#)
- [Itanium Processor Hardware Developer's Manual](#)
- Software Developer's Manual
 - [Volume 1: Application Architecture](#)
 - [Volume 2: System Architecture](#)
 - [Volume 3: Instruction Set Reference](#)
 - [Reference Manual for Software Development](#)
 - [Reference Manual for Software Optimization](#)

AMD64-Architektur:

- Programmer's Manual
 - [Volume 1: Application Programming](#)
 - [Volume 2: System Programming](#)
 - [Volume 3: General-Purpose and System Instructions](#)
 - [Volume 4: 128-Bit Media Instructions](#)
 - [Volume 5: 64-Bit Media and x87 Floating-Point Instructions](#)

 **Mailingliste**

Steht im [Wintra-System](#) zur Verfügung.

Service provided by [Mario Jeckle](#)

Generated: 2004-06-08T12:50:39+01:00

 [Feedback](#)  [SiteMap](#)

 [This page's original location: http://www.jeckle.de/vorlesung/sysarch/index.html](#)

 [RDF description for this page](#)



[Home](#) | [Products & services](#) | [Support & downloads](#) | [My account](#)

Select a country

IBM Archives > Exhibits > IBM Mainframes > Mainframes reference room >

Mainframes product profiles >

IBM Archives
Exhibits
Documents
Multimedia
Helpful links
Using the Archives site
Terms and conditions

1401 Data Processing System



Reference room areas

→ [Mainframes basic information sources](#)

→ [Mainframes product profiles](#)

→ [Mainframes photo album](#)

→ [Mainframes video views](#)

Announced October 5, 1959 and withdrawn February 8, 1971.

The following is the text of an IBM Data Processing Division press fact sheet distributed on October 5, 1959.

The all-transistorized IBM 1401 Data Processing System places the features found in electronic data processing systems at the disposal of smaller businesses, previously limited to the use of conventional punched card equipment. These features include: high speed card punching and reading, magnetic tape input and output, high speed printing, stored program, and arithmetic and logical ability.

The elements of the basic 1401 system are the 1401 Processing Unit, 1402 Card Read-Punch, and 1403 Printer. Configurations include a card system, a tape system, and a combination of the two.

The 1401 may be operated as an independent system, in conjunction with IBM punched card equipment, or as auxiliary equipment to IBM 700 or 7000 series systems.

The 1401 performs functions previously requiring a number of separate machines: card reading and punching, separation of output cards, calculating, and printing.

As an auxiliary to large scale data processing systems, the 1401 performs magnetic tape sorting and editing, card-to-tape and tape-to-card operations and high speed printing. The larger system is thus made available for data processing and logical operations.

New simplified programming techniques make the 1401 extremely powerful and more efficient than many other systems of comparable or even larger size. Variable length data and program instruction words provide maximum utilization of the magnetic core storage; there is no waste of storage capacity as with fixed record length systems. Program steps may be skipped or reread in any desired sequence, a feature which greatly increases programming flexibility.

With the 1401, manual control panel wiring is eliminated, and transfer of cards or paper between machine units is greatly reduced.

The 1401 incorporates the building block principle, providing for expansion and a variety of configurations to fit the needs of individual users.

Components of the 1401 Data Processing System

IBM 1401 Processing Unit

This unit controls the entire system by means of its stored program. It performs the arithmetic and logical functions, controls card reading and punching, magnetic tape input and output, and tells the printer what to print and where to print it.

The 1401 automatically edits the systems printed output for spacing, punctuation and format. The 1401 is available with 1,400; 2,000; or 4,000 positions of core storage. Alphabetical or numerical data may be processed.

Speed: In one minute, the 1401 Processing Unit can perform 193,300 additions (eight-digit numbers) or 25,000 multiplications (six-digit numbers by four-digit numbers).

IBM 1402 Card Read-Punch

The 1402 reads card information into the processing unit, punches cards, and separates them into radial stackers. The cards can easily be removed while the machine is running. Maximum speeds are: punching, 250 cards per minute; reading, 800 cpm. Reading and punching can be performed simultaneously.

IBM 1403 Printer

The IBM 1403 Printer is a completely new development providing maximum "thru-put" of forms and documents in printing data from punched cards and magnetic tape. The printer incorporates a swiftly moving horizontal chain (similar in appearance to a bicycle chain) of engraved type faces, operated by 132 electronically-timed hammers spaced along the printing line. The impact of a hammer presses the paper and ink ribbon against a type character, causing it to print. The chain principle achieves perfect alignment of the printed line and greatly reduces the number of sets of type characters needed.

The 1403 prints by means of a scanning operation which compares characters on the chain with characters in storage designated to be printed. When a character on the chain matches the one in storage, the hammer for that printing position is fired.

The chain of engraved type faces moves across the face of forms or documents at a constant speed of ninety inches a second. Two interchangeable

type styles are available for the chain.

An outstanding feature of the printer is the exclusive Dual Speed Carriage, which has the ability to skip over blank spaces on forms and documents at speeds far in excess of normal printing rate. This carriage skips the first eight lines at thirty-three inches per second, and beyond eight lines at seventy-five inches per second. Combined with a printing speed of 600 lines per minute, the result is a higher rate of output than is obtainable with many printers of greater line printing speed. The 1403 Printer can produce over 230 two-line documents, such as checks, per minute. This is equivalent to a printing speed of 4,800 lines per minute.

All electronic and logical operations of the printer are under the stored program control of the 1401 Processing Unit. Information to be printed is read into storage from the input cards or magnetic tape, processed, and read out to the Printer.

A feature of the Printer is a wheeled form stand which reduces and simplifies paper handling.

Magnetic Tape Units

Up to six IBM 729 Magnetic Tape Units (Model II or IV) may be added to the 1401 system for increased input, speed, and storage compactness. These are the same all-transistorized tape units used with the IBM Series 700 and 7000 data processing systems. Either single or double density tapes are specified to provide processing speeds of 15,000 or 41,667 characters a second with the 729 II; 22,500 or 62,500 characters a second for the 729 IV.

IBM 1210 Sorter-Reader

The 1401 can also be linked to an IBM 1210 Model 4 Sorter-Reader for direct processing of paper documents imprinted with magnetic ink, providing banks with a system for mechanizing bank demand deposit accounting and account reconciliation, transit operations, installment and mortgage loan accounting and other banking functions.

Typical 1401 applications

Payroll: The speed of the 1401 system is an asset in payroll processing. Payroll registers, check and earnings statements and deduction registers are prepared at speeds up to 600 lines per minute. The system will accumulate regular and overtime hours and gross earnings for each employee, check each employee's accumulated total hours and dollars, and print a daily performance record. The 1401 consolidates these payroll operations and performs them at greater speeds than present systems.

Railroad freight car accounting: The power and capacity of an IBM 1401 Tape System enables sorting, rearranging and processing of source data to be done automatically without maintaining voluminous punched card records. Source document data is recorded directly on tape, and repetitive handling and processing of records is eliminated. Greatly reduced file space, current daily information, and reduced operating expenses are among the advantages to railroad users.

Public utility customer accounting: The IBM 1401 provides the medium and smaller size public utility with an integrated operation for customer billing and accounting, starting with the application for service, continuing through the final bill, and including all records having to do with the utility customer and his premises. Some of the advantages of a 1401 Tape System to the utility are: daily updating of the entire accounts receivable file, with all cash balanced and posted on a daily basis; daily preparation of an open-item accounts receivable register, collection follow-up notices and automatic preparation of mark-sense meter reading cards as a part of the normal maintenance of the master file.

Merchandising: Decision making for merchandise control in the retail industry is a significant area for processing on the 1401. By highlighting increases in inventories and markdowns, slow-moving items can be brought to the attention of executives. Improved buying and distribution in a competitive market will be made possible by showing the effect of all transactions in a few integrated reports. Sales figures from previous weeks as well as last years figures will appear on the same reports that show the current statistics, enabling management to give attention to areas that are in difficulty. With the 1401, retail executives and merchandise managers can take advantage of faster, more accurate reporting.

Accounts receivable for retailers: The speed of an IBM 1401 Tape System and its exceptional file maintenance abilities make it well suited for handling merchandise control as well as all accounting and statistical requirements for retailing. Some of the major advantages to medium and large departments and specialty stores are tight controls to assure accuracy of customer statements, complete up-to-date credit information and ability to handle peak loads without sacrificing schedule or taking on a large temporary staff.



[Home](#) | [Products & services](#) | [Support & downloads](#) | [My account](#)

Select a country

IBM Archives > Exhibits > IBM Mainframes > Mainframes reference room >

Mainframes product profiles >

IBM Archives

Exhibits

Documents

Multimedia

Helpful links

Using the Archives site

Terms and conditions

7094 Data Processing System



Reference room areas

→ [Mainframes basic information sources](#)

→ [Mainframes product profiles](#)

→ [Mainframes photo album](#)

→ [Mainframes video views](#)

Announced January 15, 1962 and withdrawn July 14, 1969.

Built for large-scale scientific computing, the IBM 7094 Data Processing System featured outstanding price/performance and expanded computing power.

Compatible with the IBM 7090, the advanced solid-state IBM 7094 offered substantial increases in internal operating speeds and functional capacities to match growing scientific workloads in the 1960s. The powerful IBM 7094 had 1.4 to 2.4 times the internal processing speed, depending upon the individual application.

The 7094, combined with major input/output improvements through IBM 729 VI and IBM 7340 Hypertape units along with programming systems such as 7090/7094 FORTRAN, reduced job time significantly for users.

Faster Execution

The IBM 7094 achieved expanded power through high-speed processing by providing its user with

- a basic machine operating cycle of 2 microseconds
- a new processing unit which had major speed effects on:
 - floating point operations fixed point multiply and divide operations
 - index transfer instructions
 - conditional transfer instructions
 - compare operations
- two instructions per core storage cycle, substantially reducing instruction cycle time

Expanded functions

New expanded functions provided with the IBM 7094 were: double-precision floating-point operations, seven index registers, and new index-complementing instructions.

Customer conversion

The design of the IBM 7094 provided for easy conversion of the IBM 7090 data processing system to the 7094. The components and features necessary for conversion were the:

- IBM 7100 Central Processing Unit (model 2)
- IBM 7151 Console Control Unit (model 2)
- 7094 Feature (#7146) on the IBM 7606 Multiplexor

IBM customers could field convert their 7090 to a 7094 basic system in 48 to 72 system hours for minimal interruption of their activities. Existing 7090 programs using properly defined instructions could be executed without change, at increasing speeds, on the IBM 7094.

[About IBM](#) | [Privacy](#) | [Terms of use](#) | [Contact](#)



Search	FAQ	US Titles	UK Titles	Memories	VaporWare	Digest
GuestBook	Classified	Chat	Products	Featured	Technical	Museum
Downloads	Production	Fanfares	Music	Misc	Related	Contact

CED in the History of Media Technology

[Previous Image](#) | [Next Image](#) | [Slide Show Index](#) | [CED Magic Home](#)



1964: IBM Introduces the System/360 Family of Business Mainframe Computers

IBM announced their System/360 family of computers on April 7, 1964, although it was a couple years more before the system was widely available. The S/360 family consisted of six compatible computers and forty peripherals in contrast to earlier IBM computers that lost their compatibility when a new model was released. The System/360 became a mainstay of business computing for many years with IBM fulfilling orders of 1000 systems per month.

[Previous Image](#) | [Next Image](#) | [Slide Show Index](#) | [CED Magic Home](#)

----- Navigation Menu -----



▼ Resource Centers

▶ Personal Computing

Products and tools to help you buy or build the perfect PC

▶ Business/Enterprise

Find products, solutions and strategies for the enterprise

▶ Hardware Design

Tools and resources for hardware design and development needs

▶ Software Development

Get technical resources: tools, code, training, forums, & support

▶ Intel® Reseller Center

Information designed to support the growth of Intel resellers

▶ Products & Services

▶ Solutions

▶ Technologies & Trends

▶ Support & Downloads

Welcome to Your Wireless Future
There will come a time when almost every person on the planet has access to instant high-speed, anytime-anywhere Internet connections. And Intel is paving the way.

[▶ Learn More](#)

▶ Find Intel Products

[Product Support](#)

[Where to Buy](#)

[Downloads & Drivers](#)

▶ About Intel

[Press Room](#)

[Investor Relations](#)

[Governance/Social Responsibility](#)

Other Resources

Networking & Communications

Find solutions for the wired and wireless communications infrastructure & enterprises.

Small Business

See the technology & tools small or midsize businesses need for today and to plan for tomorrow.

How Intel Does IT

Read articles to learn how Intel IT plans, evaluates and deploys new technologies.

Product Highlights

Intel Processor Numbers

Intel introduces processor numbers to help end users differentiate among comparable processors.

New Embedded Microcontroller

Intel® 88CO196EC Microcontroller is a compact, cost-effective solution for embedded designs.

New Embedded Intel Processor

Get industry-leading multimedia performance from the Intel® PXA270 Processor for Embedded Computing.

News and Highlights

Fall Intel Developer Forum

Join technology and business leaders developing exciting new products through emerging technologies.

Mobile Workstation Pilot

Learn how design engineers will use wireless to work and collaborate while away from their office.

Next Generation Networks

Join our eSeminar to learn how core network infrastructure will impact the delivery of wireless services.

Ingredient #1 for Building Business



Find out how Intel technology enables business success. [See the ads.](#)

Unwire Your Life



Learn how to unwire with Intel® Centrino™ mobile technology.

INTEL 8080 Chip

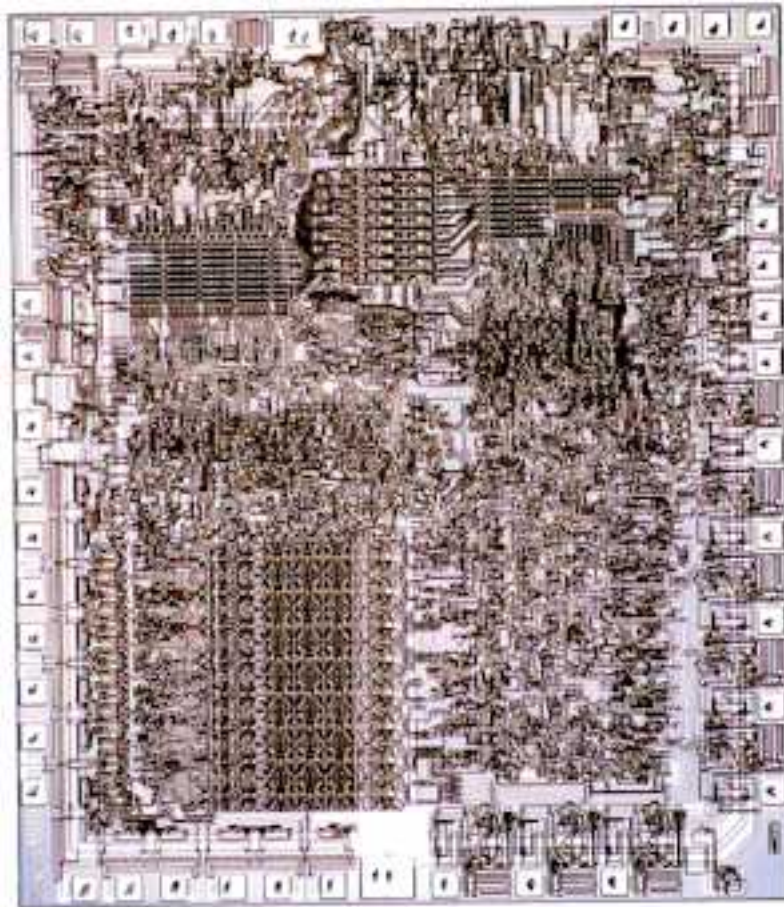


"M.Shima [21] was the 8080 project manager under Faggin. My specification used all 256 operation codes, and 12th from the bottom of my list was an obscure instruction (XTHL) for exchanging the top of stack with the HL register pair. This instruction required five memory cycles to execute, and would be used to pass arguments to subroutines. I carefully explained each instruction to Shima, whose patience was tested as I detailed the XTHL operation. He drew a line under this instruction, and declared:

"No more."

This is why the last 12 instructions were never implemented and why there was room in the instructions set for the 8085 microprocessor's added instructions [23]."

Stanley Mazor "The History of the Microcomputer - Invention and Evolution"



Intel 8080 Die - Courtesy of Stanley Mazor

The X-Number World



United States

Search

[Home](#) | [Products & services](#) | [Support & downloads](#) | [My account](#)

Select country / region



Resources for:

- Home / home office
- Small & medium business
- Large enterprise
- Government
- Education
- Developers
- IBM Business Partners
- Investors
- Journalists

[Jobs at IBM](#)[Training](#)[Administrative support](#)

[IBM TV ads](#)

THE PGA TOUR IS ON

The game of golf has entered a whole new era thanks to the power of On Demand Business.



Solutions

Solving business problems:
[IBM solutions](#) integrate hardware, software and services to meet the challenges of your industry.



News

[Engineering and technology focus of camp for 1,000 girls](#)[IBM gets 'On' new advertising campaign](#)[Five top innovators named IBM Fellows](#)[China's auto suppliers could face bumpy road](#)[→ More news and newsletters](#)

Free Port Replicator II offer

Learn more.

Services

- [Business and IT services](#)
- [Business consulting services](#)
- [On demand business](#)
- [Infrastructure services](#)
- [Financing](#)



Win a ThinkPad

Your IBM success story can win you a new ThinkPad notebook. Learn more.

Find it fast

- [Notebook finder](#)
- [Desktop finder](#)
- [Intel-based server finder](#)
- [Ready to buy?](#)
- [Special offers](#)

**ON DEMAND BUSINESS™**

Mainframes turn 40

→ From trailblazers to “dinosaurs” to business powerhouses.

[About IBM](#) | [Privacy](#) | [Terms of use](#) | [Contact](#)



CP/M Main Page

Last updated: [19 March 2004](#)

- [CP/M links](#)
- [CP/M information](#)
- [CP/M software](#)
- [ZINC - clones of some CP/M 3 utilities](#)

If you are using an Amstrad PCW, you may be interested in my [PCW page](#). If you've come here looking for PCW boot discs, you won't find them here; you need to go to [LocoScript Software](#).

What is CP/M?

It's an operating system for 8-bit computers. It looks rather like DOS to use (only not so user-friendly); this is hardly surprising because DOS was copied from CP/M in about 1980.

CP/M comes/came in three main versions; 1.4, 2.2 and 3.1. v2.2 was the basis of MSDOS, while v3.1 evolved into DRDOS and [OpenDOS / DR-DOS](#).

There were also 8086 and 68000 versions of CP/M. CP/M-86 evolved into DOS Plus, Concurrent DOS and [REAL/32](#).

Information

Articles

- A [mini-howto](#) explaining how to get DOS Plus 1.2 (CP/M-86 v4.1) running on your PC.
- [Why CP/M?](#) Dave Baldwin's USENET posting explaining his reasons for still using CP/M.
- A description of the Amstrad [PCW16](#), based on seeing it demonstrated at the Crawley PCW Club.
- [Building CP/M 3](#), thoughts on cross-compiling CP/M under DOS

- [How to read CP/M discs on a PC](#)
- ["/" as the option character in CP/M.](#)

Technical information

I hope to construct a HTML archive of all possible information about the various versions of CP/M. Some of the documents are my own; others are HTMLized versions of those at Oakland. So far, I have:

- [Disc formats](#)
- [A summary of BDOS functions](#)
- [A comprehensive description of BDOS functions](#)
- [A comprehensive description of the Graphics System eXtension](#)
- [A partial list of functions provided by RSXs](#)
- [A description of the CP/M Plus System Control Block](#)
- [The I/O Byte in CP/M 2](#)
- [A list of the CP/M 1, 2 and 3 BIOS functions](#)
- [Exact file lengths under CP/M 3](#)
- [Layout of the File Control Block](#)
- [List of BDOS replacements](#)
- [CP/M and Year 2000 issues](#)

Amstrad-specific

- [A description of the Amstrad extended BIOS](#)
- [The Amstrad device driver \(FID\) interface](#)

16-bit CP/M versions

- [How CP/M-86 version numbers relate to the BDOS version numbers](#)
- [Some information about the CP/M-86 BIOS and FIDDs](#)
- [How DOSPLUS \(CP/M-86 4.1\) implements CP/M calls on a DOS filesystem](#)
- [Why the DOSPLUS .EXE loader doesn't work, and how to fix it.](#)
- [XIOS calls provided by DOSPLUS and Personal CP/M](#)

Some file formats:

- [.COM - CP/M 3 extensions](#)
- [.CMD, the CP/M-86 executable format](#)
- [.HLP, CP/M Plus Help file format](#)
- [.LBR, Library file format](#)
- [.REL, the object file format used by the M80 and RMAC linkers](#)

- [.IRL](#), the indexed version of a .REL library
 - [.PRL](#), CP/M and MP/M binary module. Also covers SPR, RSP, RSX, ...
-

CP/M related links

What would the Web be without pages consisting entirely of links?

- USENET newsgroups:
 - [The main CP/M newsgroup](#)
 - [8-bit Amstrad computers - nearly all of these run CP/M in some shape or form.](#)
- On the Web:
 - [CP/M itself](#) and many other Digital Research products can be downloaded here.
 - [IMS Developer Documentation](#) for REAL/32 - a lot of it is relevant to earlier CP/M versions.
 - [The Computer Journal homepage](#)
 - [Discus Distribution - distributors of CP/M-86, CP/M-68k and GEM software.](#)
 - [A link page for REAL/32.](#)
 - [Herne Data Systems' CP/M page.](#)
 - [Gaby's Computermuseum and CP/M-Userclub.](#)
 - [A page dealing with Multiuser DOS for industrial applications](#)
 - [CP/Net full documentation.](#)
 - [JFRACE - CP/M in a Java applet.](#)
 - [IMSAI Series 2 - a real live CP/M computer being produced in 2002.](#)
 - [Herb Johnson's S-100 site](#)
- FTP sites:
 - [oak.oakland.edu](#), the biggest of them all. Check out the cpm, cpmug and sigm directories. The unix-c/cpm directory contains some portable tools to handle CP/M files. **Alas!** This site appears to have gone off-line owing to a disc crash.
 - [ftp.mayn.de](#) contains a mirror of most of what was at oakland, in its "archive" directory. The other directories mirror the CP/M source archives.
 - [www.retroarchive.org](#) contains some commercial CP/M software (some of which might be described as "abandonware"), and a copy of the Walnut Creek CP/M CDROM. The files from oak.oakland.edu which aren't at ftp.mayn.de can be found [on the CDROM](#).
 - [ftp.demon.co.uk](#) contains some recent Amstrad-specific stuff, some miscellaneous utilities, and ZPM3/ZCCP - a free Z-System for CP/M 3.
 - [ftp.zetnet.co.uk](#) has similar contents to Demon, but some of my PCW programs are there as well.
 - [ftp.update.uu.se](#) has some DEC Rainbow-specific stuff and MicroEmacs.
 - [soltrans.cr.usgs.gov](#) can't be reached from here, so I don't know what's in it.

- Only vaguely CP/M related:
 - [Amstrad PCW links](#).
 - [Amstrad CPC](#) web page.

[John Elliott](#) 2 Jan 2000

You are here: [About](#)>[Homework Help](#)>[Inventors](#)



Home

Essentials

[20th Century Inventions - Timelines](#)

[History of Computers/Internet](#)

[Find: A to Z Inventions](#)

[Find: A to Z Inventors](#)

[Industrial Revolution](#)

Articles & Resources

[Famous Inventions](#)

[Famous Inventors](#)

[Black Inventors](#)

[Women Inventors](#)

[Technology Timelines](#)

[Patent Trademark Copyright](#)

[Funding Licensing Marketing](#)

[Professional Inventing](#)

[Inventing 101 - Beginners](#)

[Lesson Plans, Kid Inventors](#)

[Brain Candy, Robots, Trivia](#)

[Science Clip Art](#)

[Wacky Patents and Gadgets](#)

Buyer's Guide

[Before You Buy](#)

[Top Picks](#)

[History of Television](#)

[Before You Buy An Inventors Log Book](#)

[Patent It Yourself by David Pressman](#)

Product Reviews

Articles

Forums

Help

Stay Current

Subscribe to the **About** Inventors newsletter.

Enter email address



Subscribe

Inventors

Inventors of the Modern Computer

The History of the IBM PC - International Business Machines

By [Mary Bellis](#)

In July of 1980, [IBM](#) representatives met for the first time with Microsoft's [Bill Gates](#) to talk about writing an [operating system](#) for IBM's new hush-hush "personal" computer. IBM had been observing the growing personal computer market for some time. They had already made one dismal attempt to crack the market with their [IBM 5100](#). At one point, IBM considered buying the fledgling game company Atari to commandeer Atari's early line of personal computers. However, IBM decided to stick with making their own personal computer line and developed a brand new operating system to go with. The secret plans were referred to as "Project Chess". The code name for the new computer was "Acorn". Twelve engineers, led by William C. Lowe, assembled in Boca Raton, Florida, to design and build the "Acorn". On August 12, 1981, IBM released their new computer, re-named the IBM PC. The "PC" stood for "personal computer" making IBM responsible for popularizing the term "PC".

The first IBM PC ran on a 4.77 MHz Intel 8088 microprocessor. The PC came equipped with 16 kilobytes of memory, expandable to 256k. The PC came with one or two 160k [floppy disk drives](#) and an optional color monitor. The price tag started at \$1,565, which would be nearly \$4,000 today. What really made the IBM PC different from previous IBM computers was that it was the first one built from off the shelf parts (called open architecture) and marketed by outside distributors (Sears & Roebucks and Computerland). The Intel chip was chosen because IBM had already obtained the rights to manufacture the Intel chips. IBM had used the Intel 8086 for use in its Displaywriter Intelligent Typewriter in exchange for giving Intel the rights to IBM's bubble memory technology.

Less than four months after IBM introduced the PC, Time Magazine named the computer "man of the year"

Next Chapter > [The Birth of an Operating System - MS-DOS](#)

all artwork ©MaryBellis

Subscribe to the Inventors Newsletter

Name

Email

subscribe



From [Mary Bellis](#),
Your Guide to [Inventors](#).
Sign up for my [Newsletter](#)

Search



Most Popular

[Famous Inventions](#)
[- A -](#)
[History of Inventions](#)
[Twentieth Century Inventions](#)
[1900 to 1999 - Inventors](#)
[Automobile History - The History of Cars and Engines](#)
[The History of Computers](#)
[- Computer History Timeline](#)
[Thomas Edison - The Inventions of Thomas Edison](#)

What's Hot

[The Birth of the Clothing Industry - Elias Howe and Sewing M...](#)
[License Your Patented Invention](#)
[Lewis Waterman - Fountain Pen](#)
[Asian Inventors](#)
[Henry Shrapnel - Invention of Shrapnel](#)

RATE THIS ARTICLE

Would you recommend this article?

Not at all ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 Definitely

 Submit

[Topic Index](#) |



[email to a friend](#)



font size

[back to top](#)

[Our Story](#) | [Be a Guide](#) | [Advertising Info](#) | [Investor Relations](#) | [Work at About](#) | [Help](#)

©2004 About, Inc. All rights reserved. A PRIMEDIA Company.

[User Agreement](#) | [Privacy Policy](#) | [Kids' Privacy Policy](#)



Product Families

- [Windows](#)
- [Office](#)
- [Mobile Devices](#)
- [Business Solutions](#)
- [Servers](#)
- [Developer Tools](#)
- [Games and Xbox](#)
- [MSN Services](#)
- [All Products](#)

Resources

- [Support](#)
- [Downloads](#)
- [Windows Update](#)
- [Office Update](#)
- [Learning Tools](#)
- [Communities](#)
- [Security](#)

Information For

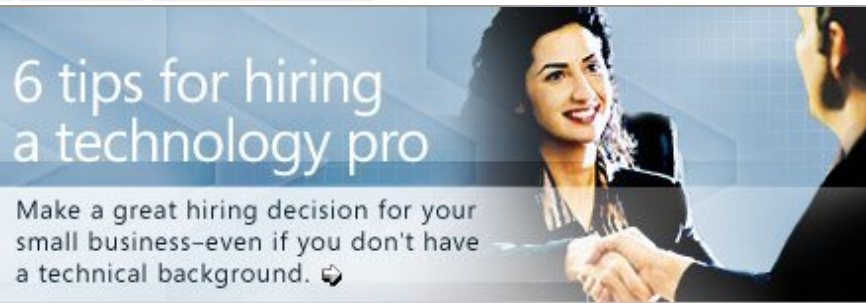
- [Home Users](#)
- [IT Professionals \(TechNet\)](#)
- [Developers \(MSDN\)](#)
- [Microsoft Partners](#)
- [Business Professionals](#)
- [Educators](#)
- [Journalists](#)

About Microsoft

- [Corporate Information](#)
- [Investor Relations](#)
- [Careers](#)
- [About this Site](#)

Worldwide

- [Microsoft Worldwide](#)



home & entertainment

- [Find out what your mouse is saying about you](#)
- [Let Windows XP back up your data so you don't have to](#)
- [Have you done the top 3 things to help protect your PC?](#)

technical resources

- [Visual Studio tools for the Microsoft Office System](#)
- [Download the SQL Server 2000 Best Practices Analyzer](#)
- [Become a Microsoft Certified Desktop Support Technician](#)

business agility

- [18 ways to have fun with Tablet PCs](#)
- [Ready for Internet phone service? 5 things to know](#)
- [Do enhanced business reporting and data analysis in Excel](#)

today's news

- [World leaders try out Tablet PCs at G8 Summit](#)

More Microsoft News ...

popular downloads

- [Microsoft Office 2000 Service Pack 3](#)
- [Internet Explorer 6 Service Pack 1 update](#)
- [Windows Media Player 9 Series](#)
- [Cumulative security update for Microsoft Outlook Express 6 Service Pack 1](#)

More Downloads ...

popular searches

- [Sasser](#)
- [Worm](#)
- [Movie Maker](#)
- [Internet Explorer](#)

More Searches ...

support

- [Downloads and updates](#)
- [Service packs for Microsoft products](#)
- [3 steps to help ensure your PC is protected](#)

Windows

- [Internet Explorer: How and why to clear your cache](#)
- [How Windows can easily help you fix a mistake](#)
- [Choose from 4 editions of Microsoft Windows XP](#)

More Windows ...

Office

- [See what programs are in Office 2003 Editions](#)
- [7 mobile services that work with Office](#)
- [Access work e-mail messages on any computer with an Internet connection](#)

More Office ...

Windows Server System

- [Download Exchange Server 2003 SP1](#)
- [What do you want in "Longhorn" Server?](#)
- [Free learning plan: Assess your Windows Server System skills](#)

More Windows Server System ...

Visual Studio .NET

- [Top 10 MSDN code samples](#)
- [Watch online movies to improve your Visual Basic skills](#)
- [XML support in SQL Server 2005](#)

Microsoft Business Solutions

- [7 wonders of automated payroll](#)
- [Create reports that people really read](#)
- [How to figure the total cost of a business solution](#)

MSN

- [Use MSN Messenger to keep your life organized](#)
- [Download MSN Messenger 6.2](#)
- [Get one-click access to e-mail: Download the MSN Toolbar](#)

[**More Visual Studio ...**](#)

[**More Microsoft
Business
Solutions ...**](#)

[**More MSN ...**](#)

• [**Use Office Update to
scan your computer
and install updates**](#)

[**More Searches ...**](#)

Last Updated: Monday, June 7, 2004 - 11:02 a.m. Pacific Time

[Manage Your Profile](#) | [Contact Us](#) | [Microsoft This Week! Newsletter](#) | [Legal](#)

© 2004 Microsoft Corporation. All rights reserved. [Terms of Use](#) | [Trademarks](#) | [Privacy Statement](#)

You are here: [About](#)>[Homework Help](#)>[Inventors](#)



Home

Essentials

[20th Century Inventions - Timelines](#)

[History of Computers/Internet](#)

[Find: A to Z Inventions](#)

[Find: A to Z Inventors](#)

[Industrial Revolution](#)

Articles & Resources

[Famous Inventions](#)

[Famous Inventors](#)

[Black Inventors](#)

[Women Inventors](#)

[Technology Timelines](#)

[Patent Trademark Copyright](#)

[Funding Licensing Marketing](#)

[Professional Inventing](#)

[Inventing 101 - Beginners](#)

[Lesson Plans, Kid Inventors](#)

[Brain Candy, Robots, Trivia](#)

[Science Clip Art](#)

[Wacky Patents and Gadgets](#)

Buyer's Guide

[Before You Buy](#)

[Top Picks](#)

[History of Television](#)

[Before You Buy An Inventors Log Book](#)

[Patent It Yourself by David Pressman](#)

Product Reviews

[Articles](#)

[Forums](#)

[Help](#)

Inventors

Inventors of the Modern Computer

The History of the MS-DOS Operating Systems, Microsoft, Tim Paterson, and Gary Kildall

Search



IBM PC

Inventors of the Modern Computer Series

• [Table of Contents](#)

• [Next Chapter](#)

The Apple Lisa and Consumer GUI

[ENTER](#)

More on MS-DOS, Time Paterson, and Gary Kildall

• [A Short History of MS-DOS](#)

Written by Tim Paterson himself.

• [Origins of MS-DOS](#)

Articles about MS-DOS and Tim Paterson.

• [The Deal of the Century](#)

More on Microsoft's purchase of the "Quick and Dirty Operating System".

• [Microsoft Timeline](#)

• [Windows Operating](#)

[Systems Family History](#)

From

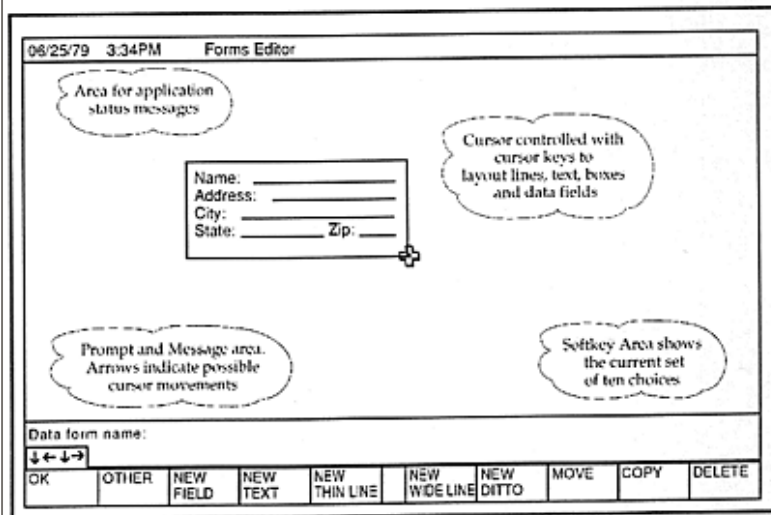
Stay Current

Subscribe to the **About** Inventors newsletter.



Subscribe

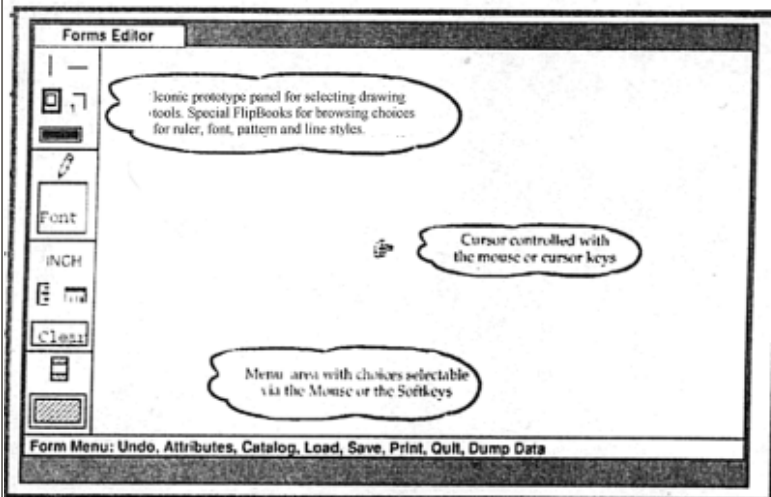
A Pictorial History of the "Apple Desktop Interface" 1979 - 2000



July 1979 Prototype

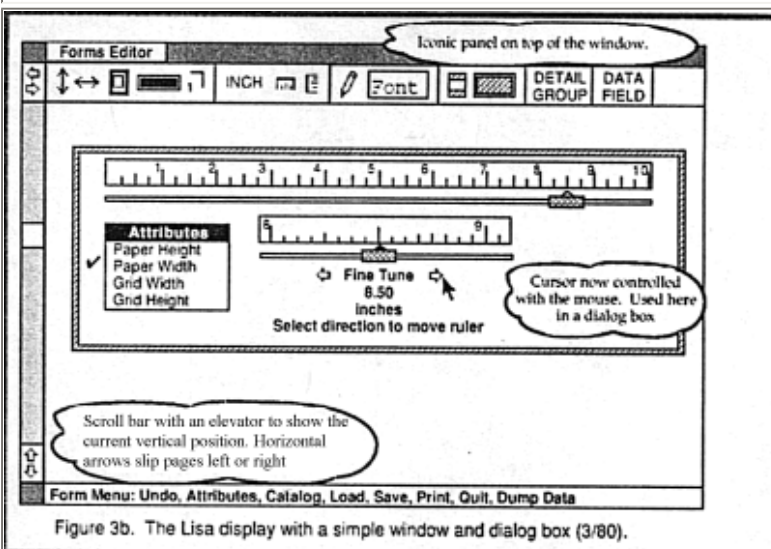
(Developed on an Apple] [)

- Area for application status messages
- Cursor controlled with cursor keys to layout lines, text, boxes and data fields.
- Prompt and Message areas. Arrows indicate possible cursor movements.
- Softkey Area shows the current set of ten choices.



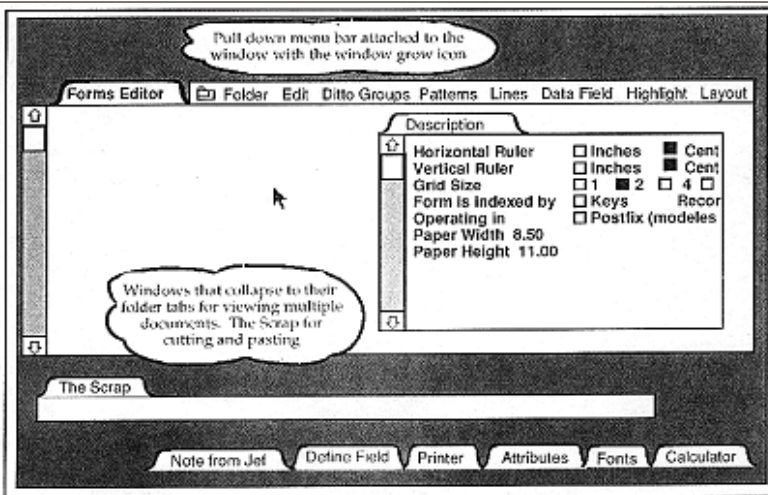
February 1980 Prototype

- Iconic prototype panel for selecting drawing tools. Special FlipBooks for browsing choices for ruler, font, pattern and line styles.
- Cursor controlled with the mouse or cursor keys.
- Menu area with choices selectable via the Mouse or the Softkeys.



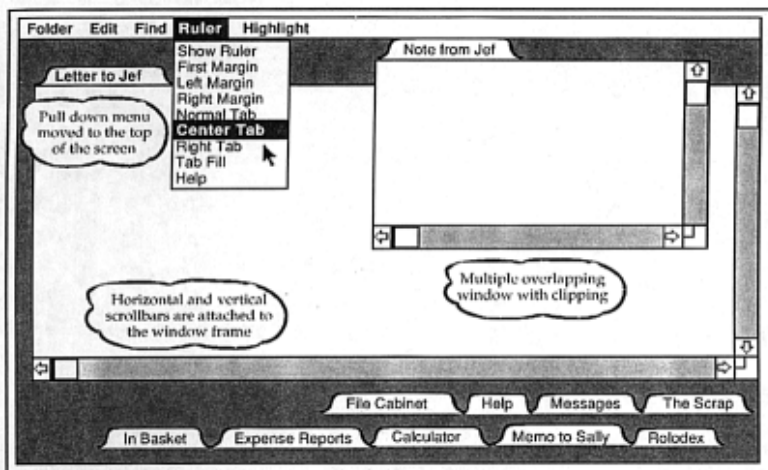
March 1980 Prototype

- Iconic panel on top of the window.
- Cursor now controlled with the mouse. Used here in a dialog box.
- Scrollbar with an elevator to show the current vertical position. Horizontal arrows slip pages left or right.



August 1980 Prototype

- Pull down menu bar attached to the window with the window grow icon.
- Windows that collapse to their folder tabs for viewing multiple documents. The Scrap for cutting and pasting.



October 1980 Prototype

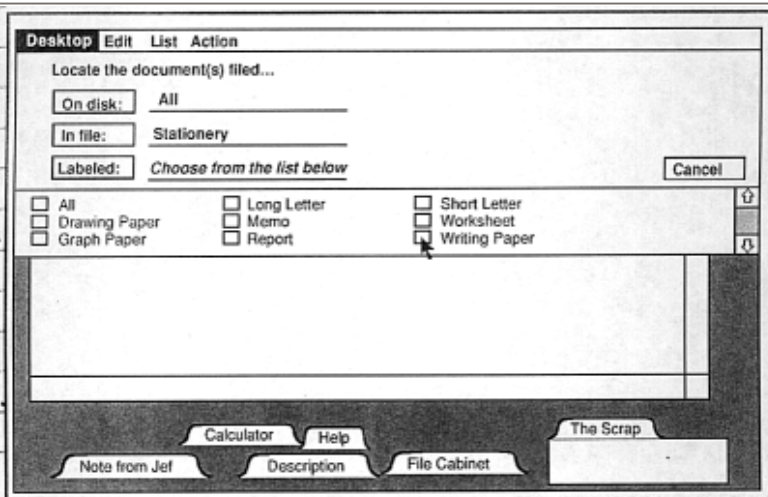
- Pull down menu moved to the top of the screen.
- Horizontal and vertical scrollbars are attached to the window frame.
- Multiple overlapping window with clipping.



December 1980 Prototype

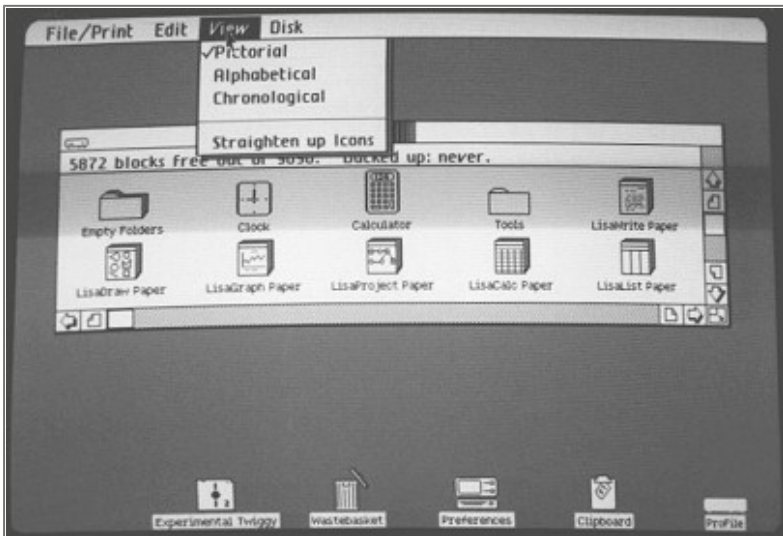
- Browser-like display with individually resizable panes.

(Looks a lot like the NeXT-style "Browser View" coming in [Mac OS X](#), doesn't it?)



July 1981 Prototype

(Notice the dialog slides out from under the menubar. Remind of you the new feature in [Mac OS X](#)?)



Lisa Office System 1.0

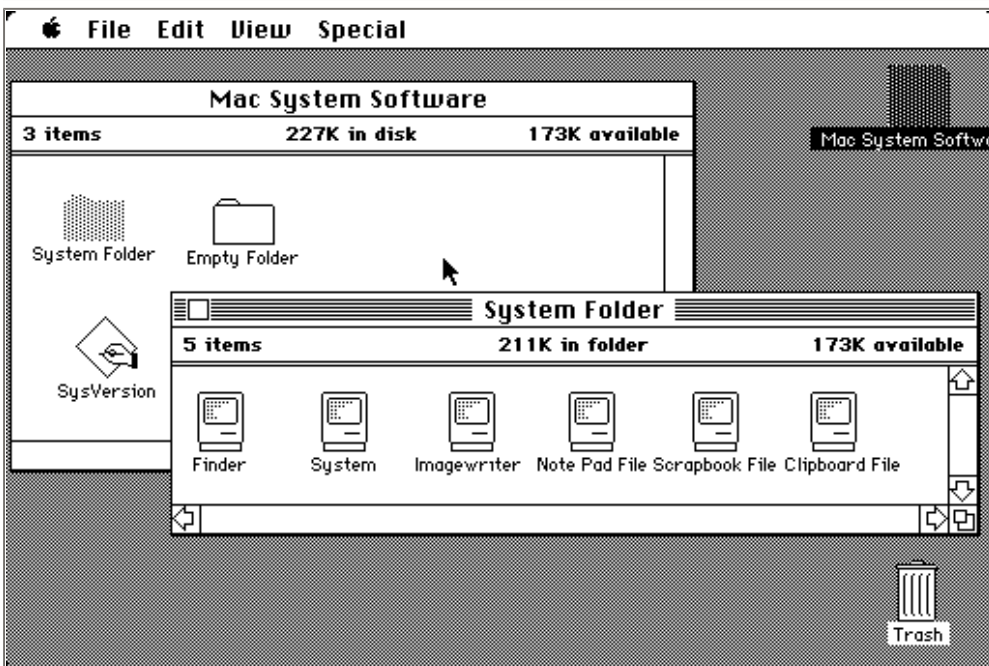
(May 1983)



Lisa 7/7 Office System 3.1

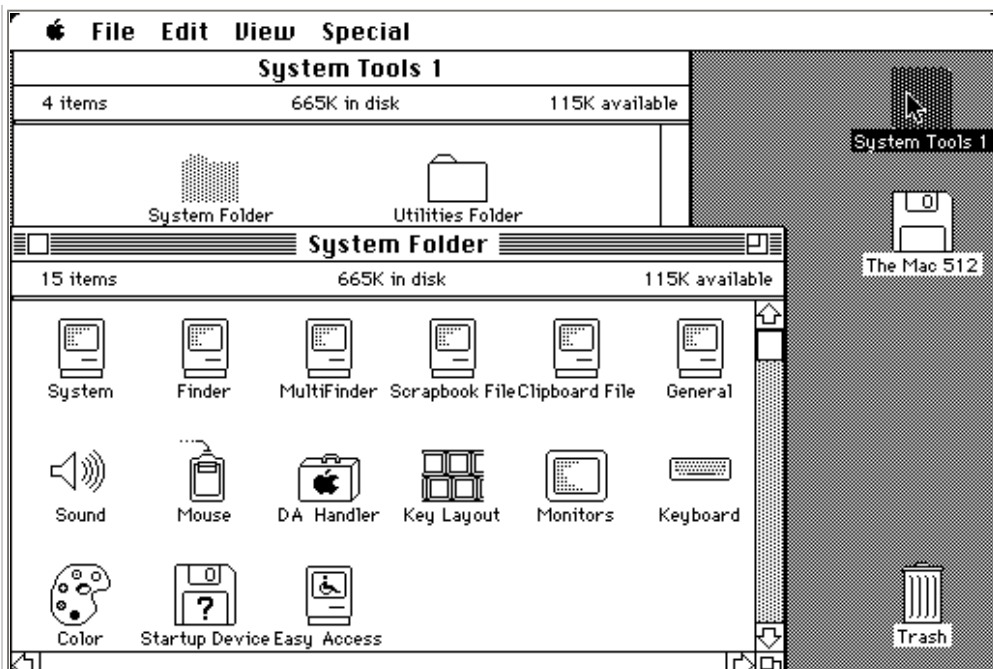
(1984, Final Release)

(Notice the Desk menu holding both running apps and Desktop items with open items indicated by a checkmark. Lisa Office System was a cooperative multitasking environment with protected memory.)



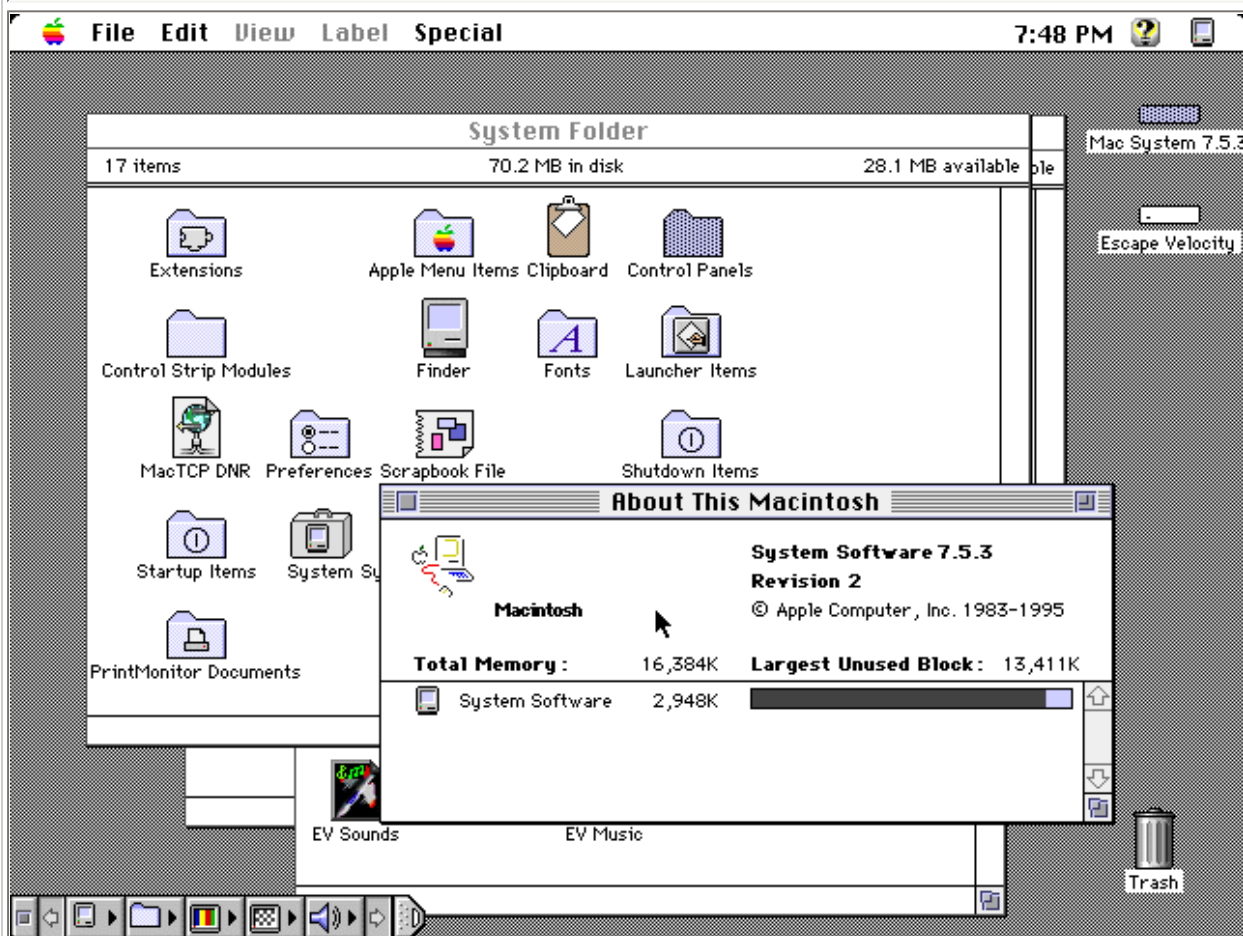
January 1984

- Macintosh System 1.0.



1987

- Macintosh System 4.2 (Finder version 6)



1995

- Macintosh System 7.5.3

Summer
2000: [Mac OS X](#)



(January
2000
MacWorld
Keynote)

"Prototype" snapshots Copyright© 1997 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page or initial screen of the document. Take from *Interactions*, vol 4, issue 1, pp 40-53.

"Lisa Office System 1.0" and "Lisa 7/7 Office System 3.1" photos Copyright© 1998 Tom Stepleton. [Apple Lisa Web page](#)

"Mac OS X" screenshot Copyright© 2000 [Apple](#).


[about parc](#)
[research](#)
[events](#)
[contact us](#)

The Palo Alto Research Center (PARC), a subsidiary of Xerox Corporation, conducts pioneering interdisciplinary research in physical, computational, and social sciences. Building on our three-decade tradition of innovation, PARC works with [Xerox](#) and other strategic partners to commercialize technologies created by our renowned scientists.

As the birthplace of technologies such as laser printing, Ethernet, the graphical user interface, and ubiquitous computing, PARC has an established track record for transforming industries and creating commercial value.

NEWS FROM PARC

To receive updates on PARC, please [subscribe](#) to our email newsletter.

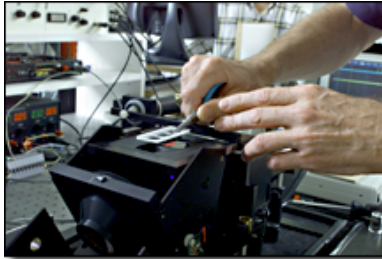
[PARC Campus an archetype for Silicon Valley](#)

Outside blends with the environment while the inside fosters creative egalitarianism.

[Cancer detection with a laser](#)

Scripps-PARC has developed a technique using lasers that can detect the cancer cells in as little as two minutes—as much as 1,000 times faster than digital microscopy.

[news archive](#)



The Scripps-PARC Institute for Advanced Biomedical Sciences unveils new method aimed at early cancer detection

[Press Release, April 19, 2004](#)

In March, PARC unveiled technologies that address two critical challenges in ubiquitous computing: security and interoperability.

- [Making Security Usable](#)
- [Securing Wireless Networks](#)
- [Obje™ Interoperability Platform](#)
- [Press Release, March 2, 2004](#)



Using PARC's prototype system, users were able to set up 802.1x networks using digital certificates in less than two minutes.

UPCOMING EVENTS

June 8, 2004

BayCHI In Search of Efficient Text Writing Methods for Off-Desktop Computing: An Overview of ATOMIK and SHARK

June 10, 2004

[PARC Forum: Computer Networks & Packet Switch Design](#)

June 10, 2004

Bay Area MEMS Journal Club Monthly Meeting

[information for upcoming events](#)

[RSS 1.0](#)
[RSS 2.0](#)
[ABOUT](#)

info@parc.com

[privacy policy](#)

Linus Torvalds (torvalds@transmeta.com)

The homepage of a WWW-illiterate

Why does this exist at all?

Frankly, I don't know. I got a default homepage (in Finnish) made automatically for me, and now I wonder what I should do with it. If you have any great suggestions, feel free to mail me, and I'll probably feel free to ignore you.

If you're looking for **Linux** information, you'll find more of it somewhere else, because I'm hopeless when it comes to documentation. You could try [Lasu's Linux Index](#) for starters

Linux v2.0

This is Patricia Miranda Torvalds, my daughter:



Linux Logo

See [here](#) for the penguin logos I like. I'm including an animated version here too,



Contact Information

Snail-mail (work):

Linus Torvalds
Transmeta Corp
3940 Freedom Circle
Santa Clara, CA 95054
USA

Phone (work):

+1 (408) 327 9830 x328

Email:

torvalds@transmeta.com

Herman Hollerith

aus Wikipedia, der freien Enzyklopädie

Herman Hollerith (* 29. Februar 1860, † 17. November 1929 in Washington, D.C.) war ein US-amerikanischer Unternehmer, Ingenieur und Erfinder. In Buffalo im US Bundesstaat New York als Kind deutscher Einwanderer aus der Pfalz geboren studierte er an der Columbia University Ingenieurwissenschaften. Hollerith war graduerter Bergwerksingenieur und hatte Patente für eine automatische Eisenbahn-Druckluftbremse und für die maschinelle Herstellung von Wellblechplatten angemeldet.

Da er zunächst als Statistiker für die amerikanische Regierung arbeitete, beschäftigte er sich mit der Erfassung und Speicherung von Daten mittels Lochkarten. Als Vorbild verwandte er ein zur damaligen Zeit im Eisenbahnbereich gebräuchliches System, das mittels mehrerer Löcher in den Fahrkarten die Fahrgäste nach Geschlecht und Alter klassifizierte. Er entwickelte ein System zur Erfassung von Daten auf Lochkarten. Am 8. Januar 1889 meldet er sein System zum Patent an. Er wird damit zum Begründer der maschinellen Datenverarbeitung. Sein System wurde bei der Volkszählung 1890 angewandt und trug zu einer starken Beschleunigung der Auszählung bei.

Nach weiteren Verbesserungen des Systems gründete er schließlich 1896 die *Tabulating Machine Company* um seine Erfindung kommerziell zu verwerten. 1911 fusionierte die *Tabulating Machine Company* mit der *Computing Scale Corporation* und der *International Time Recording Company* zur *Computing Tabulating Recording Corporation* (CTR). 1924 wurde CTR schliesslich in *International Business Machines Corporation* (IBM) umbenannt.

-
- Diese Seite wurde zuletzt geändert um 10:40, 24. Mär 2004.
 - Der Inhalt dieser Seite steht unter der GNU Free Documentation License.



▼ aktuell
science

suchmaschine subscribe forum impressum



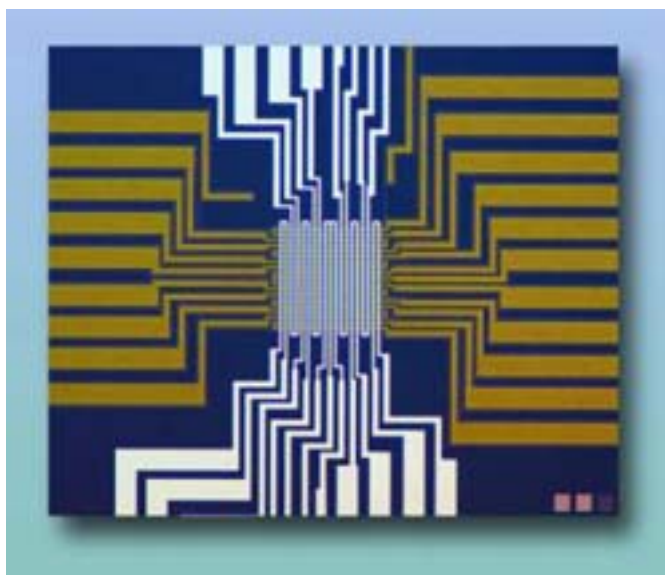
Auf der Suche nach dem Heiligen Gral in der Welt der Halbleiterspeicher

Andrea Naica-Loebell 01.08.2002

Verbesserung der MRAM-Speichertechnik

In der aktuellen Ausgabe des Wissenschaftsmagazins [Nature](#) wird von einem neuen Verfahren zur Verbesserung der MRAM-Speichertechnik berichtet. Diese ultraschnelle magnetoelektronische Schaltung könnte einen Durchbruch für magnetische Arbeitsspeicher bedeuten.

download



MRAM-Chip von IBM

 [weitere artikel](#)

[Eistanz in Richtung Spin-Elektronik](#)

[Coole Halbleiter](#)

[Halbleiter-Industrie in der Krise](#)

MRAM ist die Kurzform für *Magnetic Random-Access Memory*, in Deutsch Hochintegrierter magnetischer Festkörperspeicher genannt. Diese neue Speichertechnologie könnte die Computer revolutionieren. Wer heute nicht ständig auf die Festplatte speichert, verliert seine Daten, sobald der Arbeitsspeicher nicht mehr mit Strom versorgt wird. Im Moment werden dynamische Schreib- und Lesespeicher verwendet (DRAM, Dynamic Random-Access Memory), bei denen die elektrische Ladung aber immer wieder aufgefrischt werden muss, um die Daten nicht zu verlieren. Dieser so genannte Refresh-Vorgang kostet Zeit und birgt immer das Risiko, dass bei einem Zusammenbruch der Versorgungsspannung alle aktuellen Inhalte weg sind. MRAMs bewahren die Information ohne ständige Stromzufuhr. Werden sie verwendet, entfällt auch das zeitaufwändige Laden der Programme beim Start des Computers. Booten in Nullzeit könnte die Zukunft sein.

Das ist auch und besonders für mobile Anwendungen, also Laptops, Organizer oder digitale Film- und Fotokameras interessant. MRAM ist sehr schnell, kostengünstig und erreicht die gleiche Packungsdichte wie DRAM, da sie via Mikrostrukturierung als Festkörperspeicher aufgebaut ist (Vgl.  [Gigabit-Speicher mit Langzeitgedächtnis](#)).

Kein Wunder, dass das Interesse der Industrie an der Forschung sehr hoch ist. MRAM-Bausteine werden seit geraumer Zeit bereits in der Raumfahrt verwendet. Das Bundesministerium für Bildung und Forschung fördert das Leitprojekt  [Magnetoelektronik](#) und diverse Konzerne beteiligen sich mit umfassenden finanziellen und personellen Ressourcen. IBM (Vgl.  [MagRAM](#)) und Infineon (Vgl.  [IBM und Infineon forcieren Entwicklung von revolutionärer Speichertechnologie](#)) sind bei der wissenschaftlichen Jagd auf die neue Technologie dabei und wollen bis 2004 entsprechende magnetische Speicherbausteine bis zur Marktreife gebracht haben.

Im Juni 2002 präsentierte Motorola einen 1-Mbit-MRAM-Chip, der ab 2004 in die Massenproduktion gehen soll. Der 256-Kbit-MRAM-Chip wird bereits ausgeliefert. Jim Handy, Direktor für den Bereich nicht-flüchtige Speicher, sagte anlässlich der Präsentation:

► Forscher haben jahrzehntelang nach dem 'heiligen Gral' in der Welt der Halbleiterspeicher gesucht -- einen Baustein, der nicht-flüchtig, kostengünstig, schnell und stromsparend ist. Jede der heute gebräuchlichen Technologien, ob DRAM, Flash oder SRAM, adressiert eine oder zwei dieser Eigenschaften, aber keine Technologie kann bisher alle vier Anforderungen erfüllen. Die Erfolge von Motorola auf dem Gebiet der MRAM-Technologie scheinen uns näher an das ultimative Ziel eines idealen Speicherchips zu bringen.

(Vgl. [Motorola erreicht bedeutenden Durchbruch mit einem 1 Mbit MRAM-Universal-Speicherchip mit Kupfermetallisierung](#)). Der potenzielle weltweite Markt für Arbeitsspeicher wird auf mindestens 30 Milliarden Dollar jährlich geschätzt.

Th. Gerrits, H.A.M. van den Berg, J. Holfeld und Th. Rasing vom Research Institute for Materials der holländischen [Universität Nijmegen](#) und L. Bär von der [Siemens AG](#) präsentieren jetzt in Nature ihre Verbesserung der MRAM-Technologie. In MRAM-Arbeitsspeichern werden die Bits magnetisch kodiert. Dabei haben sie einen ähnlichen Aufbau wie die historischen Ringkernspeicher (Vgl. [Aufbau der Z22 Hardware](#)), wobei sich an der Stelle der Ringkerne so genannte Tunnelkontakte befinden, die aus zwei magnetischen Schichten bestehen, die durch eine Isolatorschicht getrennt sind.

Die Ausrichtung der Magnetisierung in den beiden Schichten bestimmt die Kodierung der Bits, antiparallel für 0 und parallel für 1. Die Magnetfeldorientierungen der Zellen werden durch das Anlegen eines äußeren Magnetfelds dirigiert, das bestimmt, wann von 0 auf 1 umgeschaltet wird, damit Daten gespeichert werden können. Dem Team um Gerrits ist es gelungen, die Geschwindigkeit und Verlässlichkeit der magnetischen Rotation (Vgl. [Präzession](#)) durch gesteuerte magnetische Feldimpulse (*shaped magnetic field pulses*) zu steigern und das [Wellenphänomen](#) auszuschalten, das normalerweise dafür sorgt, dass die Magnetisierung nach einer schnellen Rotation in ihren Ausgangszustand zurückfällt.

Dadurch ist es möglich, ultraschnell eine Ummagnetisierung zu

erreichen, das heißt, die Magnetisierung wird gezielt gegeneinander ausgerichtet und wie gewünscht umgeschaltet. Der Impuls wird durch die zeitversetzte Überlagerung von zwei Laserimpulsen in Pikosekunden erreicht (eine Pikosekunde = 0,000000000001 Sekunde).



 [artikel versenden](#)

Kommentare:

[Re: Bevor man dummes Zeug von heutigen OSs redet,besser informieren \(Buchtipp\).](#)
[\(dmoorhuhn, 26.9.2002 18:42\)](#)

[also los und einschreiben \(flowerp, 16.9.2002 9:07\)](#)

[Bevor man dummes Zeug von heutigen OSs redet,besser informieren \(Buchtipp\). \(Onkel](#)
[Wanja, 2.8.2002 11:10\)](#)

[mehr...](#)

[↑ top](#)

Copyright © 1996-2002. All Rights Reserved. Alle Rechte vorbehalten

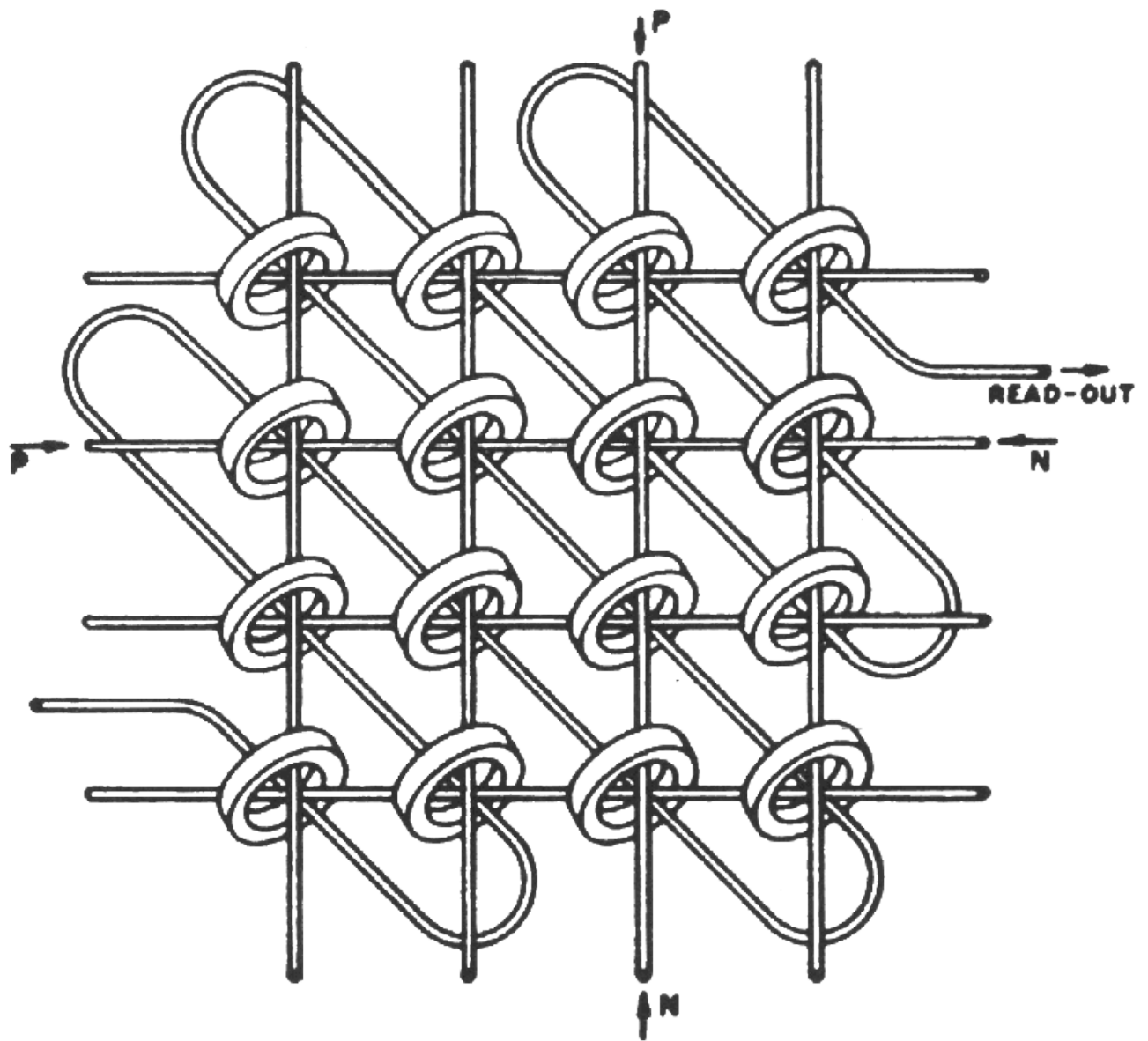
Verlag Heinz Heise, Hannover

last modified: 31.07.2002

[Privacy Policy / Datenschutzhinweis](#)



redaktion



Simulation

aus Wikipedia, der freien Enzyklopädie

Unter **Simulation** versteht man die Nachbildung von Prozessen oder Situationen in einem Modell, um daraus gezielt Erkenntnisse zu ziehen. Dabei konzentriert man sich auf die Aspekte des simulierten Systems, die für die Erkenntnisgewinnung von besonderem Interesse sind. Andere Aspekte des simulierten Systems hingegen, die für die zu beantwortende Fragestellung (vermutlich) nur eine geringe Rolle spielen, werden in der Simulation vereinfacht oder weggelassen. Im Zusammenhang mit Simulation spricht man von dem zu simulierenden System und von einem Simulationsmodell, welches eine Abstraktion des zu simulierenden Systems darstellt.

Ein Auto-Crashtest beispielsweise ist ein Simulationsmodell für eine reale Verkehrssituation, in der ein Auto in einen Verkehrsunfall verwickelt ist. Dabei wird die Vorgeschichte des Unfalls, die Verkehrssituation und die genaue Beschaffenheit des Unfallgegners stark vereinfacht. Auch werden keine Personen in den simulierten Unfall involviert, sondern es werden stattdessen Crashtest-Dummies eingesetzt, die mit realen Menschen gewisse mechanische Eigenschaften gemeinsam haben. Ein Simulationsmodell hat also nur ganz bestimmte Aspekte mit einem realen Unfall gemeinsam. Welche Aspekte dies sind, hängt maßgeblich von der Fragestellung ab, die mit der Simulation beantwortet werden soll.

Für den Einsatz von Simulationen kann es mehrere Gründe geben:

- Eine Untersuchung am realen System wäre zu aufwändig, zu teuer oder zu gefährlich. Beispiele:
 - Crashtest (zu gefährlich in der Realität)
 - Simulation von Fertigungsanlagen vor einem Umbau (mehrfacher Umbau der Anlage in der Realität wäre zu aufwändig und zu teuer)
- Das reale System existiert (noch) nicht. Beispiel: Windkanalexperimente mit Flugzeugmodellen, bevor das Flugzeug gefertigt wird
- Das reale System lässt sich nicht direkt beobachten
 - Systembedingt. Beispiel: Simulation einzelner Moleküle in einer Flüssigkeit
 - Das reale System arbeitet zu schnell. Beispiel: Simulation von Schaltkreisen

- Das reale System arbeitet zu langsam. Beispiel: Simulation geologischer Prozesse
- Für Experimente kann ein Simulationsmodell wesentlich leichter modifiziert werden, als das reale System. Beispiel: Modellbau in der Stadtplanung
- Gefahrlose und kostengünstige Ausbildung. Beispiel: Flugsimulation
- Spiel und Spaß an simulierten Szenarien

Heutzutage werden Simulationen mehr und mehr durch Computer realisiert, weil Computer ein ideales und sehr flexibles Umfeld für fast alle Arten der Simulation bieten (siehe auch Computersimulation).

Inhaltsverzeichnis

1 Typen und Bereiche von Simulationen

1.1 Typen von Simulationen

2 Theorie, Modell und Simulation

3 Weblinks

Typen und Bereiche von Simulationen

Grundsätzlich muss man zwischen Simulationen mit und ohne Computer unterscheiden. Eine Simulation ist ein "Als ob"-Durchspielen von Prozessen; das kann man auch ohne Computer tun. Wenn heute von "Simulation" die Rede ist, meint man allerdings fast immer **Computersimulationen**. Letztere gliedern sich in die Bereiche

1. Spielsimulationen, z.B. Flugsimulatoren, Wirtschaftssimulationen
2. Technische Simulationen, z.B. Schaltungssimulationen, Atomwaffensimulationen, Windkanalsimulationen u.v.m.
3. Wissenschaftliche Simulationen. Sie gibt es in fast allen Natur- und Gesellschaftswissenschaften:
 - Metereologische Simulation zur Wettervorhersage
 - Physikalische Simulation und astrophysikalische Simulation
 - Chemische Simulation
 - Biologische Simulationen, z.B. Simulation neuronaler Netze (siehe neuronales Netz)
 - Sozioökonomische Simulation, z.B. Multiagentensysteme
 - u.v.m.

Typen von Simulationen

Viel zu tun...

Hier sei nur erwähnt:

- Makrosimulation und Mikrosimulation
- Materialfluss Simulation (DES Discrete Material Flow Simulation)
- Statistische Simulation, insbesondere Monte Carlo-Simulation
- Numerische Simulation

Theorie, Modell und Simulation

Eine Computersimulation besteht im Kern aus einem Programm. Hier werden die Regeln der Prozesse definiert. Man spricht analog zu den "Verhaltensgleichungen" eines mathematisch definierten Systems von den "Verhaltensalgorithmen". Ihre Gesamtheit stellt das Modell dar, das die Simulation *realisiert*, falls diese Simulation die Realität abbilden soll. (Was z.B. bei Spielsimulationen i.a. nicht im Vordergrund steht). Modelle können ad hoc erfunden werden, sie können aber auch aus empirischen Befunden oder aus einer Theorie abgeleitet werden. Z.B. werden die Wettermodelle aus der Theorie der Hydrodynamik abgeleitet, die einzelnen Verhaltensgleichungen bestehen aus Navier Stokes-Gleichungen (oder Weiterentwicklungen davon).

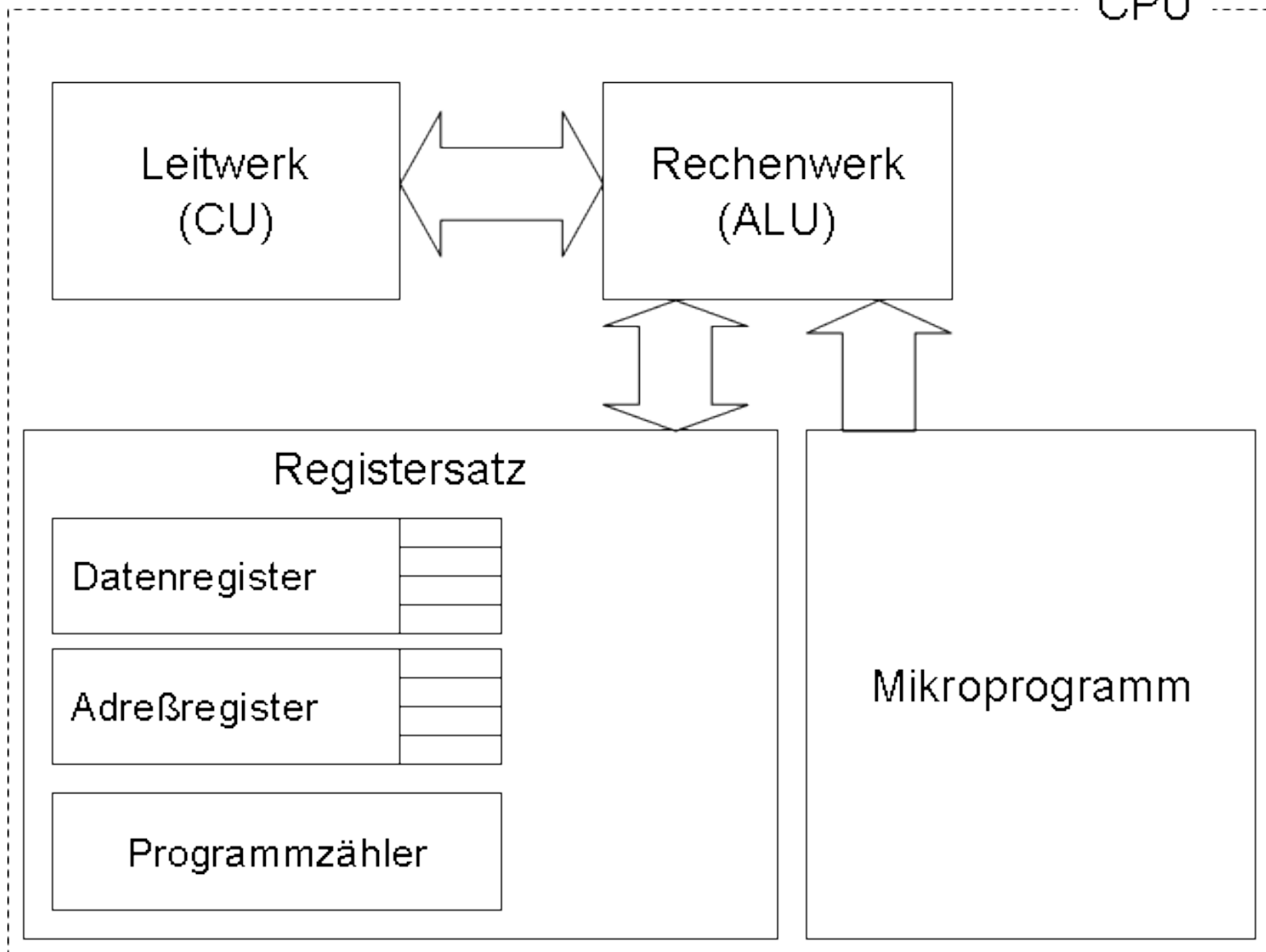
Weblinks

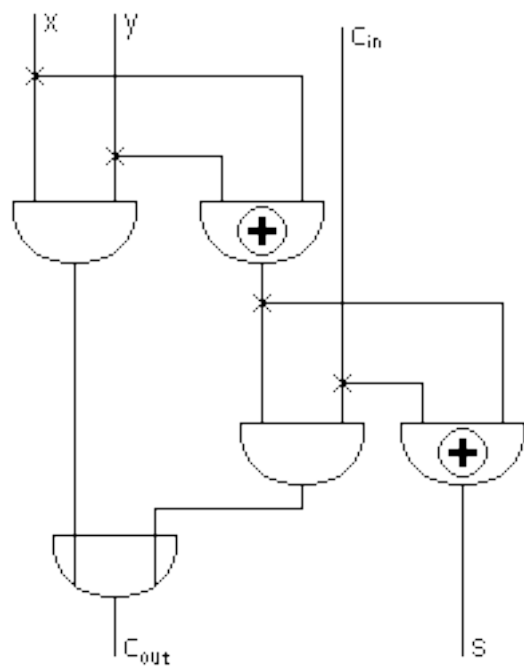
- <http://www.soc.surrey.ac.uk/research/simsoc/> CRESS - Center for Research in Social Simulation
- <http://www.askos.de> Büro für Analyse und Simulation komplexer Systeme
- <http://www.envi-met.com> Mikroklimatisches, dreidimensionales Modell

Siehe auch: Emulator

-
- Diese Seite wurde zuletzt geändert um 14:01, 14. Mai 2004.
 -

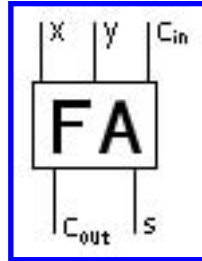
Der Inhalt dieser Seite steht unter der [GNU Free Documentation License](#).





Volladdierer

aus Wikipedia, der freien Enzyklopädie



Schaltsymbol

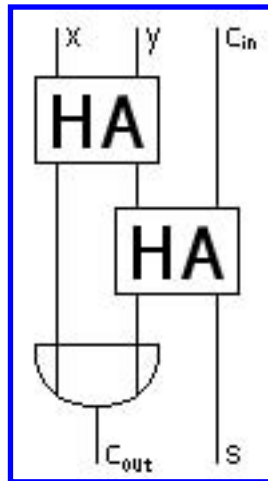
Ein **Volladdierer** (engl. full adder) ist ein elektronischer Baustein mit drei Eingängen und zwei Ausgängen. Mit einem Volladdierer kann man drei einstellige Binärzahlen addieren. Dabei liefert der Ausgang s (engl. sum - *Summe*) die rechte Stelle des Ergebnisses, der Ausgang c_{out} (engl. carry (output) - *Übertrag (Ausgang)*) die linke.

Da Volladdierer meist so hintereinandergeschaltet werden, dass der Ausgang c_{out} eines Volladdierers mit einem Eingang des nächsten verbunden ist, wird dieser Eingang hier c_{in} genannt.

Die folgende Wahrheitstafel zeigt die Funktionsweise eines Volladdierers:

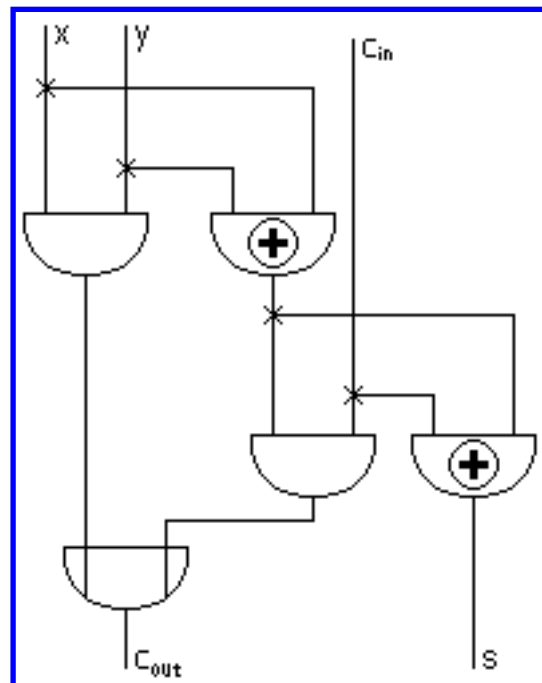
x	y	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0

1	1	0	1	0
1	1	1	1	1



Aufbau Vollsubtrahiere aus Halbaddierern

Die rechte Abbildung zeigt den Aufbau eines Volladdierers mittels Halbaddierern und einem Oder-Gatter.

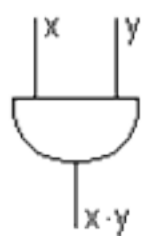


Aufbau Volladdierer

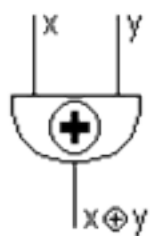
Die linke Abbildung zeigt ebenfalls den Aufbau eines Volladdierers, wobei die Halbaddierer jeweils in ein Und-Gatter und ein XOR-Gatter aufgetrennt wurden.

Der Volladdierer wird zum Aufbau von Addiernetzen verwendet, oft in Kombination mit einem Halbaddierer.

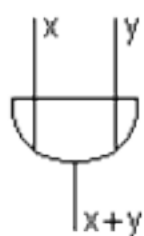
-
- Diese Seite wurde zuletzt geändert um 23:57, 14. Mär 2004.
 - Der Inhalt dieser Seite steht unter der GNU Free Documentation License.



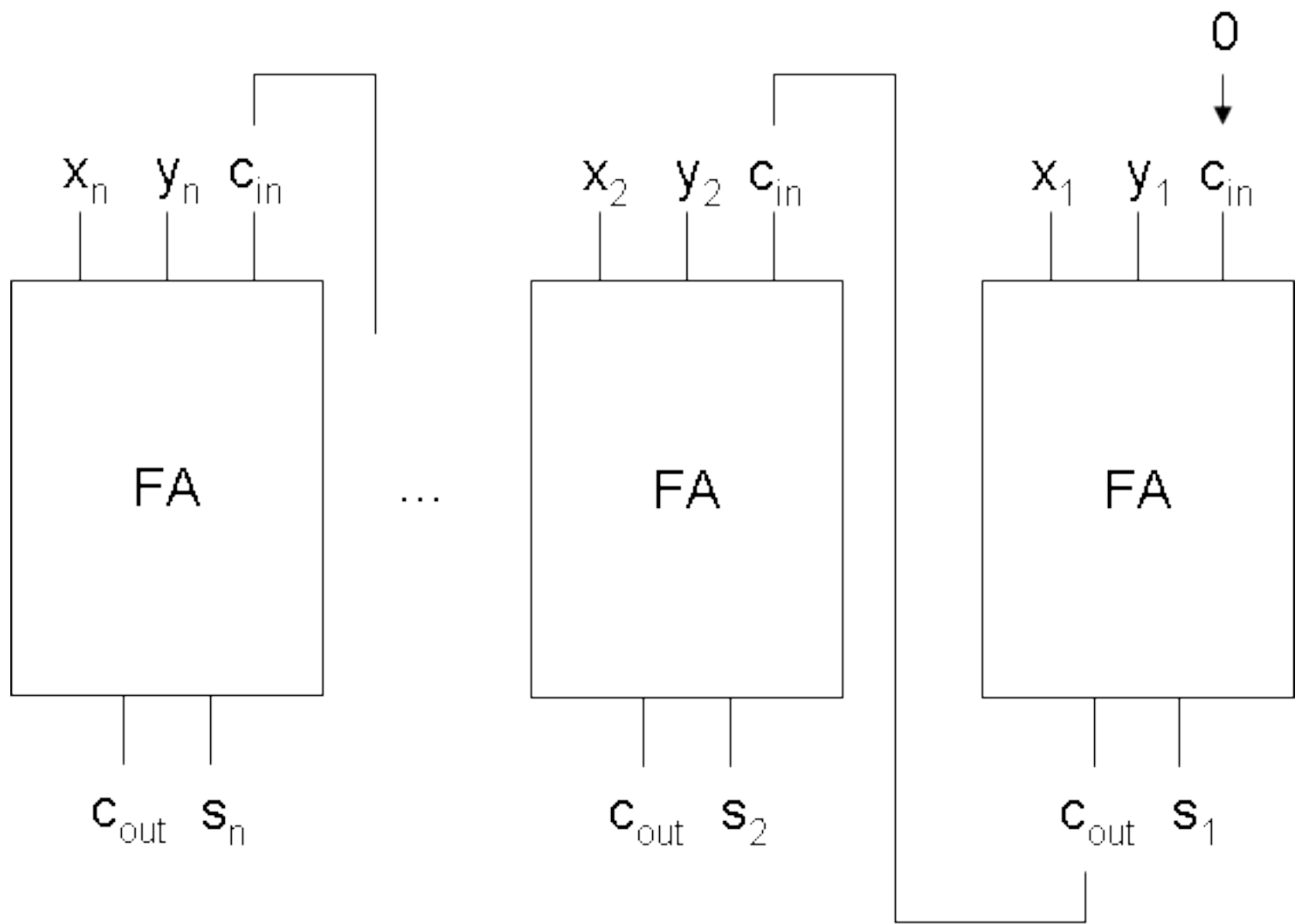
AND

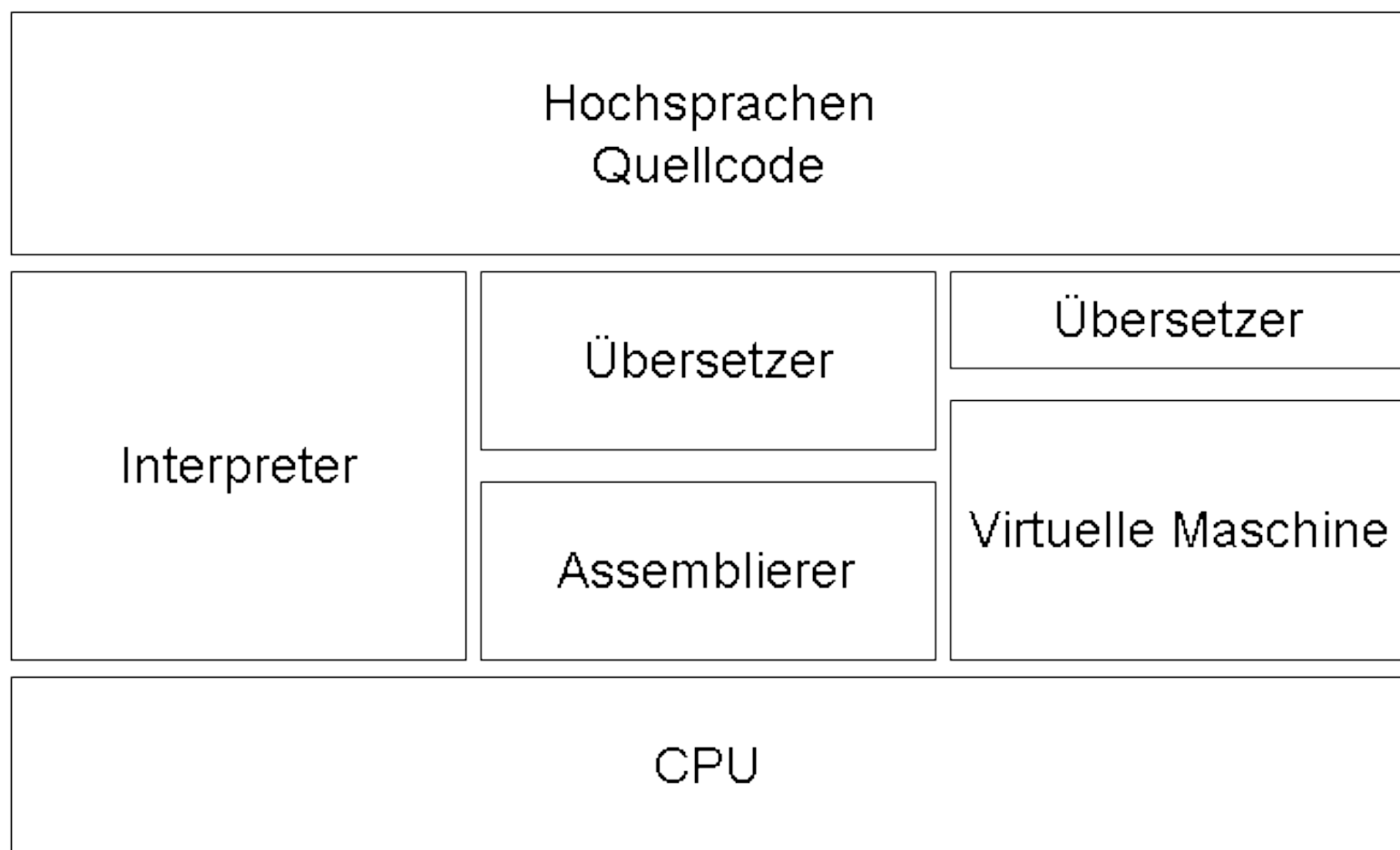


XOR

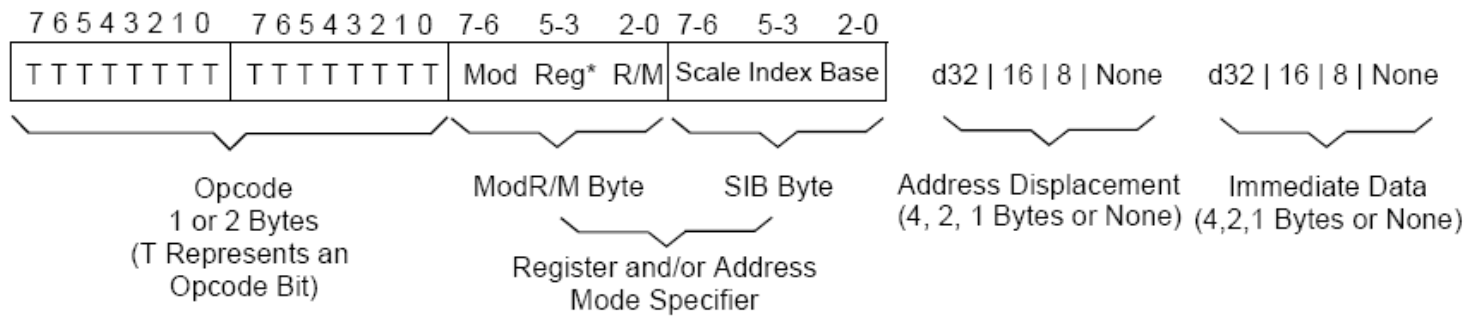


OR





[illegible]



* Reg Field is sometimes used as an opcode extension field (TTT).

The Pentium FDIV bug - a Picture

Intel Pentium chips manufactured before a certain date have a bug in their floating point processor which returns less than full precision results for some combinations of divisor and dividend when performing a Floating point DIVision (FDIV). A well known operation which exhibits this phenomenon is 4195835/3145727. You can check to see if your Pentium has the FDIV bug by entering the following formula in the Windows calculator:

$$(4195835 / 3145727) * 3145727 - 4195835$$

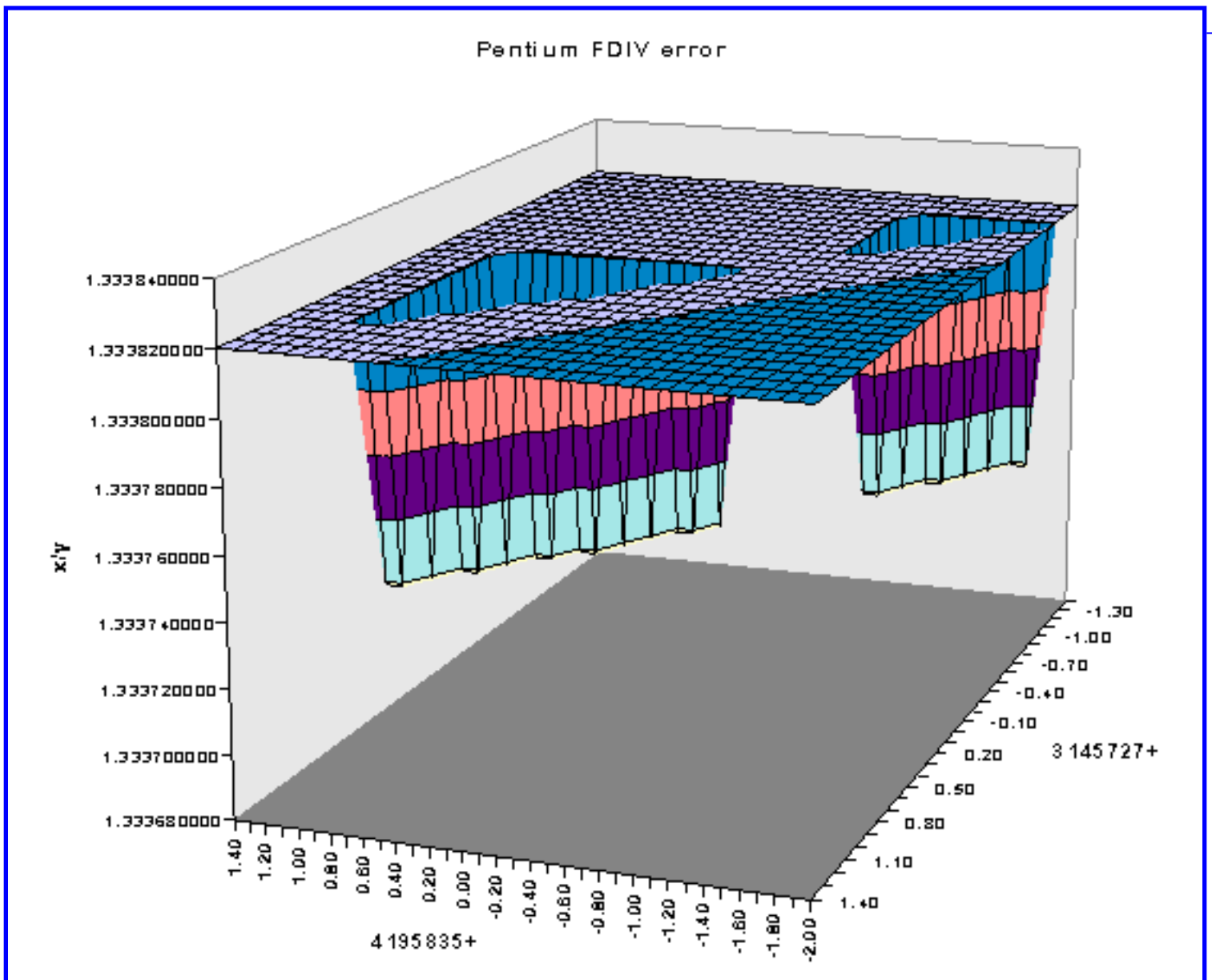
The image below is a 3d graph of the function x/y in the region of 4195835/3145727. Specifically, the region is:

$$4195833.0 \leq x \leq 4195836.4$$

$$3145725.7 \leq y \leq 3145728.4$$

On a 486/66, the function graphs as a monotonically increasing surface with the low point in the foreground corner (where x is low and y is high) and a high point in the background corner (where x is high and y is low). On a Pentium with the FDIV bug there are two triangular areas in the region where an incorrect result is returned. The correct values all would round to 1.3338 and the incorrect values all would round to 1.3337, an error in the 5th significant digit.

An archive of principal papers on the Pentium FDIV bug is available at: <http://www.mathworks.com/company/pentium/index.shtml>



A ZIP file containing the Microsoft Excel spreadsheet along with an Encapsulated Postscript copy of this graphic can be retrieved via: [anonymous ftp](#)

Larry Hoyle (lhoyle@stat1.cc.ukans.edu)
[Institute for Public Policy and Business Research](#)
[University of Kansas](#)

Nov. 30, 1994

[▶ Resource Centers](#)[▶ Products & Services](#)[▶ Solutions](#)[▶ Technologies & Trends](#)[▶ Support & Downloads](#)
[Search](#)

Pentium® Processors

Statistical Analysis of Floating Point Flaw

Intel White Paper

End of Interactive Support Announcement

These products are no longer being manufactured by Intel. Additionally, Intel no longer provides interactive support for these products via telephone or e-mail, nor will Intel provide any future software updates to support new operating systems or improve compatibility with third party devices and software products.

THESE DOCUMENTS ARE PROVIDED FOR HISTORICAL REFERENCE PURPOSES ONLY AND ARE SUBJECT TO THE TERMS SET FORTH IN THE "LEGAL INFORMATION" LINK BELOW. For information on currently available Intel products, please see www.intel.com and/or developer.intel.com

Intel Corporation

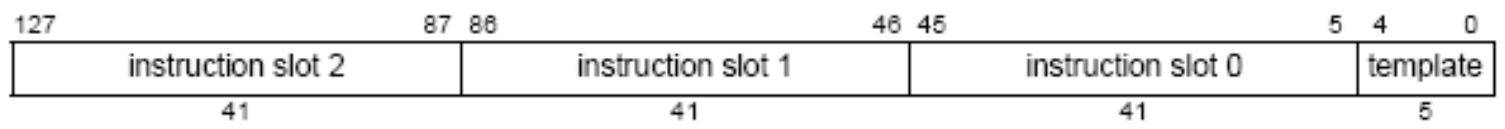
November 30, 1994

Table of Contents

- [Section 1: Abstract](#)
- [Section 2: Introduction](#)
- [Section 3: Description of the Flaw](#)
- [Section 4: Pentium® Processor Divide Algorithm](#)
- [Section 5: Evaluation Framework to Gauge Impact on User](#)
- [Section 6: Analysis of Impact on Applications](#)
- [Section 7: Conclusions](#)
- [Section 8: Acknowledgments](#)
- [Section 9: References](#)
- [Appendix A](#)

[Site Index](#) | [*Legal Information](#) | [Privacy Policy](#) | [Contact Us](#)

©2004 Intel Corporation





UNISYS

Unisys History Newsletter

The [Unisys](#) History Newsletter is written and published by [George Gray](#). George is a Systems Programmer for the **State of Georgia Department of Administrative Services** and is heavily involved in [Unite Inc.](#), a Unisys User Group. He began his work on the Unisys History Newsletter as a hobby and privately published his first six newsletters. Then, he began writing a regular computer history column in [UniSphere magazine](#). There are a number of articles from a period of time (1994-1998) that are not available to us due to copyright issues. Hopefully, one day UniSphere will put those articles online or give us permission to publish them here. Fortunate for us, George continues to research and record computer history and make the newer articles available for all to enjoy.

With George's permission, I am able to bring you these fascinating articles on-line.

Enjoy!

- [UNIVAC in Pittsburgh 1953-1963](#), Vol. 1, Num. 1 (September 1992, revised 1999).
- [The UNIVAC Solid State Computer](#), Vol. 1, Num. 2 (December 1992, revised 1999).
- [EXEC II](#), Vol. 1, Num. 3 (March 1993, revised 1999).
- [The UNIVAC 1100 in the Early 70s](#), Vol. 1, Num. 4 (June 1993, revised 1999).
- [The UNIVAC III Computer](#), Vol. 2, Num. 1 (September 1993, revised 1999).
- [The UNIVAC File Computer](#), Vol. 2, Num. 2 (December 1993, revised 1999).
- [Some Burroughs Transistor Computers](#), Vol. 3, Num. 1 (March 1999).
- [The UNIVAC 1108](#), Vol. 3, Num. 2 (April 1999).
- [Engineering Research Associates and the Atlas Computer \(UNIVAC 1101\)](#), Vol. 3, Num. e (June 1999).
- [Sperry Rand Military Computers 1957-1975](#), Vol. 3, Num. 4 (August 1999).
- [Burroughs Third-Generation Computers](#), Vol. 3, Num. 5 (October 1999).
- [Remington Rand Tabulating Machines](#), Vol. 4, Num. 1 (May 2000).
- [The UNIVAC 418 Computer](#), Vol. 4, Num. 2 (August 2000, revised May 2003).
- [UNIVAC 1: The First Mass-Produced Computer](#), Vol. 5, Num. 1 (January 2001).
- [The Burroughs 220 Computer](#), Vol. 5, Num. 2 (April 2001).
- [UNIVAC I Instruction Set](#), Vol. 5, Num. 3 (June 2001).
- [The UNIVAC 1102, 1103, and 1104](#), Vol. 6, Num. 1 (January 2002).
- [UNIVAC and ALGOL](#), Vol. 6, Num. 2 (June 2002).

You may also be interested in [other articles by George Gray](#).

Other Resources

Check out Philly.com's March 29, 2001 interview of George Gray about [UNIVAC's 50th Anniversary](#). You might also be interested in reading the ensuing discussion on the [related Slashdot thread](#).

You can find more information about the history of computing at these other Internet sites:

- [IEEE Annals of the History of Computing](#)
- [IEEE History Center](#)
- [John W. Mauchly and the Development of the ENIAC Computer](#)
- [Mauchly: The Computer and the Skateboard \(Real/Quicktime video\)](#)
- [UNIVAC Memories](#), by John Walker.
- [Rowaton Historical Society](#), Remington Rand history, etc.
- [About.com's ENIAC/UNIVAC links](#).
- [PBS's "Triumph of the Nerds"](#)
- [Smithsonian](#) Computer History.
- [Charles Babbage Institute](#) - dedicated to the preservation of the history of information processing.
- [Jones Multimedia Encyclopedia](#) - Computers: History and Development.
- [Yahoo Directory](#) - Computers and Internet: History.
- Mike Muuss' [Computer History Information](#).
- [Computer History Museum](#).
- [Virtual Museum of Computing History](#).
- [Sperry Calculators!?](#)
- [Book: A History of Modern Computing](#),
[Book Review](#).
- [Chronology - The BIG List](#).
- [Doug Engelbart's 1968 Demo](#).
- [Bob Bemer \(pioneer recalls computer history\)](#).

Also, you can find more information about Unisys at the [UNITE](#), the Unisys User's Association.



[Randy Carpenter](#)

Georgia Institute of Technology
College of Computing
Atlanta, GA 30332-0280
randy@cc.gatech.edu

You are here: [About](#)>[Homework Help](#)>[Inventors](#)



Home

Essentials

[20th Century Inventions - Timelines](#)

[History of Computers/Internet](#)

[Find: A to Z Inventions](#)

[Find: A to Z Inventors](#)

[Industrial Revolution](#)

Articles & Resources

[Famous Inventions](#)

[Famous Inventors](#)

[Black Inventors](#)

[Women Inventors](#)

[Technology Timelines](#)

[Patent Trademark Copyright](#)

[Funding Licensing Marketing](#)

[Professional Inventing](#)

[Inventing 101 - Beginners](#)

[Lesson Plans, Kid Inventors](#)

[Brain Candy, Robots, Trivia](#)

[Science Clip Art](#)

[Wacky Patents and Gadgets](#)

Buyer's Guide

[Before You Buy](#)

[Top Picks](#)

[History of Television](#)

[Before You Buy An Inventors Log Book](#)

[Patent It Yourself by David Pressman](#)

Product Reviews

Articles

Forums

Help

Stay Current

Subscribe to the **About** Inventors newsletter.



Subscribe

Inventors

Inventors of the Modern Computer

The History of the UNIVAC Computer - J. Presper Eckert and John Mauchly

By [Mary Bellis](#)

The Universal Automatic Computer or UNIVAC was a computer milestone achieved by Dr. Presper Eckert and Dr. John Mauchly, the team that invented the [ENIAC](#) computer.



J Presper Eckert and John Mauchly, after leaving the academic environment of The Moore School of Engineering to start their own computer business, found their first client was the United States Census Bureau. The Bureau needed a new computer to deal with the exploding U.S. population (the beginning of the famous baby boom). In April 1946, a \$300,000 deposit was given to Eckert and Mauchly for the research into a new computer called the UNIVAC.

The research for the project proceeded badly, and it was not until 1948 that the actual design and contract was finalized. The Census Bureau's ceiling for the project was \$400,000. J Presper Eckert and John Mauchly were prepared to absorb any overrun in costs in hopes of recouping from future service contracts, but the economics of the situation brought the inventors to the edge of bankruptcy.

In 1950, Eckert and Mauchly were bailed out of financial trouble by Remington Rand Inc. (manufacturers of electric razors), and the "Eckert-Mauchly Computer Corporation" became the "Univac Division of Remington Rand." Remington Rand's lawyers unsuccessfully tried to re-negotiate the government contract for additional money. Under threat of legal action, however, Remington Rand had no choice but to complete the UNIVAC at the original price.

On March 31, 1951, the Census Bureau accepted delivery of the first UNIVAC computer. The final cost of constructing the first UNIVAC was close to one million dollars. Forty-six UNIVAC computers were built for both government and business uses. Remington Rand became the first American manufacturers of a commercial computer system. Their first non-government contract was for General Electric's Appliance Park facility in Louisville, Kentucky, who used the UNIVAC computer for a payroll application.

Inventors of the Modern Computer Series

• [Table of Contents](#)

• **Next Chapter**

International Business

Machines

IBM 701 EDPM

[ENTER](#)

UNIVAC Computer

More on the UNIVAC Computer - J. Presper Eckert and John Mauchly

• [Further Reading:](#)

The Unisys History Newsletter- six articles covering the history and technical data of the UNIVAC. The Paul Revere of Computers - an excellent essay on the business history of the UNIVAC. The UNIVAC flow chart and pictures of the UNIVAC computer.

Most Popular

[Famous Inventions](#)

[- A -](#)

[History of Inventions](#)

[Twentieth Century](#)

[Inventions](#)

[1900 to](#)

[1999 -](#)

[Inventors](#)

[Automobile](#)

[History -](#)

[The](#)

[History of](#)

[Cars and](#)

[Engines](#)

[The](#)

[History of](#)

[Computers](#)

[- Computer](#)

[History](#)

[Timeline](#)

[Thomas](#)

[Edison -](#)

[The](#)

[Inventions](#)

[of Thomas](#)

[Edison](#)

What's Hot

[The Birth of the](#)

[Clothing](#)

[Industry -](#)

[Elias Howe](#)

[and](#)

[Sewing M...](#)

[License](#)

[Your](#)

[Patented](#)

[Invention](#)

[Lewis](#)

[Waterman](#)

[- Fountain](#)

[Pen](#)

[Asian](#)

[Inventors](#)

[Henry](#)

[Shrapnel -](#)

[Invention](#)

[of Shrapnel](#)

UNIVAC SPECS

The UNIVAC had an add time of 120 microseconds, multiply time of 1,800 microseconds and a divide time of 3,600 microseconds.

Input consisted of magnetic tape with a speed of 12,800 characters per second with a read-in speed of 100 inches per second, records at 20 characters per inch, records at 50 characters per inch, card to tape converter 240 cards per minute, 80 column punched card input 120 characters per inch, and punched paper tape to magnetic tape converter 200 characters a second.

Output media/speed was magnetic tape/12,800 characters per second, uniprimer/10-11 characters per second, high speed printer/600 lines per minute, tape to card converter/120 cards per minute, Rad Lab buffer storage/Hg 3,500 microsecond, or 60 words per minute.

J Presper Eckert and John Mauchly's UNIVAC was a direct competitor with IBM's computing equipment for the business market. The speed with which UNIVAC's magnetic tape could input data was faster than IBM's [punch card](#) technology, but it was not until the presidential election of 1952 that the public accepted the UNIVAC's abilities.

In a publicity stunt, the UNIVAC computer was used to predict the results of the Eisenhower-Stevenson presidential race. The computer had correctly predicted that Eisenhower would win, but the news media decided to blackout the computer's prediction and declared that the UNIVAC had been stumped. When the truth was revealed, it was considered amazing that a computer could do what political forecasters could not, and the UNIVAC quickly became a household name. The original UNIVAC now sits in the Smithsonian Institution.

Next Chapter > [The History of International Business Machines - IBM 701 EDPM](#)

artwork@marybellis

Subscribe to the Inventors Newsletter

Name

Email



From [Mary Bellis](#),
Your Guide to [Inventors](#).
Sign up for my [Newsletter](#)

RATE THIS ARTICLE

Would you recommend this article?

Not at all ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 Definitely

[Topic Index](#) |

About



[email to a friend](#)



font size

[back to top](#)

[Our Story](#) | [Be a Guide](#) | [Advertising Info](#) | [Investor Relations](#) | [Work at About](#) | [Help](#)

©2004 About, Inc. All rights reserved. A PRIMEDIA Company.

[User Agreement](#) | [Privacy Policy](#) | [Kids' Privacy Policy](#)

UNIVAC[®] Memories

This document is an ever growing collection of memorabilia, contemporary documents, and anecdotes recounting the history of UNIVAC 1100 series mainframes. The first computer I ever used was a UNIVAC 1107, and for more than a decade stretching from 1967 through 1978, most of my programming was oriented toward those machines, spanning four generations of hardware: the 1107, 1108, 1110, and 1100/80 (which I used briefly to develop microprocessor software).

As the collection grows, I will organise it more coherently and make it easier to find your way around. In these early days of the archive, it's simply a collection of unrelated documents linked to the items below.

Enjoy, and may [minus zero](#) never be (completely) forgotten.

[The UNIVAC 1107 at Case Institute of Technology](#)

In September of 1967 I wrote my first computer program and ran it on this machine. This illustrated document recalls the characteristics of a supercomputer, 1960's style.

[The FASTRAND Page](#)

People who have used a 100 megabyte hard drive that weighed two and a quarter tons and cost more than US\$130,000 in 1968 experience a special sense of wonder when tucking one of today's 2.1 gigabyte drives, just purchased for less than US\$1000 and weighing less than half a kilo, into their pocket. [The FASTRAND Page](#) turns the clock back to the days when mass storage was massive.

[Typical UNIVAC 1108 Prices in 1968](#)

In 1968 you could pick up a 1.3 MHz CPU with half a megabyte of RAM and 100 megabyte hard drive for a mere US\$1.6 million. Oh, and you want a printer too...?

[UNIVAC 1107/1108/1110 Instruction Set](#)

What UNIVAC programmer can ever forget the day he or she found a way to use an instruction like "Magnitude of Characteristic Difference to Upper"? Here are all the instructions for the 1100 family from the 1107 to 1110, colour-coded to indicate which machine included what instructions, with SLEUTH I and SLEUTH II mnemonics.

[Minus Zero](#)

UNIVAC 1100 machines used one's complement representation of negative numbers, as opposed to the two's complement form almost universal today. In one's complement, where zero is a *signed number*, you have not only zero, but *minus zero* to contend with. This document explores various representations of negative numbers and describes some of the bizarre consequences and applications of minus zero.

[Derek Zave's Amazing Octal Editor](#)

Nothing better illustrates what could be accomplished with the rather odd UNIVAC 1100 instruction set than this program, which edits a 36-bit number into a 12 digit octal string in exactly 12 instructions, with no loops.

[Cute Tricks on the UNIVAC 1100](#)

Programmers of UNIVAC 1100 mainframes in the 1960's through the mid 1970's exalted in discovering clever ways to squeeze the most out of machines which, by today's standards, would be considered hopelessly slow and short of memory. Here's a collection of assembly language and FORTRAN coding tricks for the UNIVAC 1100 collected in 1974 and 1975 by Walter Gilbert of the University of Maryland Administrative Computer Center.

[Chi Corporation Code Card](#)

If they can put a man on the Moon, why can't they put a UNIVAC 1108 in your shirt pocket? Well, a few months *before* a man walked on the Moon, programmers at Chi Corporation in Cleveland did the next best thing--a shirt-pocket reference card for the UNIVAC 1108 packed with essential information for assembly language systems programmers. Can't find yours after all these years? Here's your own personal copy, on the Web.

[The FIELDATA Character Code](#)

Back in the '60s, almost every mainframe computer had its own character code. The UNIVAC 1100 machines used a 6-bit variant of FIELDATA code, originally designed by the [U.S. Army](#).

[Punch Card Gallery](#)

Programmers of the mainframe era tended to be better typists than those who started with timesharing terminals or personal computers. Why? Because the only way to get your program into the computer was to punch it on cards, and as you discovered within seconds after sitting down at the keypunch, "you can't erase a hole". Since the computer only reads the holes, cards could be any colour and bear any design, and many sites printed their own custom cards, sometimes quite elaborate. Here's a gallery of punch cards from UNIVAC sites, including a rare 90-column card dating from the era when 80-column "IBM cards" were the exclusive property of IBM.

[UNIVACKY](#)

This hilarious mid-1970's poem by Harry Gilbert, in the spirit of Lewis Carroll's [Jabberwocky](#), packs more EXEC-8 acronyms than you can probably recall into a delightful story which can't help but summon memories of the late night "code rage" which occasionally possessed even the most mild-mannered UNIVAC system programmers.

[The Morse Code Exec](#)

In 1971, John Walker figured out how to convert a multi-million dollar room-sized UNIVAC 1108 mainframe into a Morse *code practice oscillator* with a stand-alone, bootable operating system dedicated to that dubious application. Here it is, complete with source code.

[The ANIMAL Episode](#)

- [ANIMAL Source Code](#)
- [PERVADE Source Code](#)

Ever heard the one about the self-reproducing ANIMAL program that spread throughout systems and colonised new machines? I wrote it. Here's the story, both as it actually happened, and recounted in the press as a "computer urban legend" 10 and 15 years after the fact.

[FANG Source Code Browser](#)

Freeware, 1972-style! FANG was one of the rare applications to fully exploit the multi-tasking, multi-processor, and multiple I/O channel architecture of the UNIVAC 1100 series. A file and tape utility, it allowed simultaneous execution of any number of commands, while guaranteeing results identical to the much slower serial execution employed by UNIVAC's own utilities. FANG was freely available, first released in 1972, and was used by hundreds of UNIVAC sites. It was last modified in 1975, and remains in use today, more than 20 years later. To view this document, you need a browser which supports frames.

UNIVAC has been, over the years, a registered trademark of Eckert-Mauchly Computer Corporation, Remington Rand Corporation, Sperry Rand Corporation, Sperry Corporation, and [Unisys Corporation](#). FASTRAND is a trademark of Sperry Rand Corporation, since merged into [Unisys Corporation](#).

[Other Computing History Resources at Fourmilab](#)

[Fourmilab Home Page](#)

[by John Walker](#)

IBM 1401

```
,008015,022029,036043,050054,055062,063065,066077/333/M0762502F1.  
HELLO  
WORLD
```

This is a self loading card for the IBM 1401 that you run by putting the card in the reader and hitting START.
The program will print HELLO WORLD, eject the page, and halt.

submitted by: tom.vanvleck@Eng.Sun.COM (Tom Van Vleck)



Home

History	People	Code	Sites	Stories	About

[main site](#)[Stratus mirror](#)[Brisbane mirror](#)

Multics Reunion and Dinner Honoring Professor Fernando J. Corbató

Professor John V. Guttag, head of MIT's Department of Electrical Engineering and Computer Science (EECS), cordially invites all Multics alumni/ae to MIT's new Stata Center on Saturday, June 19, 2004. [\[details\]](#) (06/03/04, 6K) **updated**

Multics (Multiplexed Information and Computing Service) is a mainframe timesharing operating system begun in 1965 and used until 2000. Multics began as a research project and was an important influence on operating system development. The system became a commercial product sold by Honeywell to education, government, and industry. This web site describes the hardware, software, and people that made the system the best thing of its kind for many years.

Professor Corbató sent a [letter to all Multicians](#) on the occasion of the last system's shutdown on 10/31/2000.

General Information and FAQ

General information about Multics and info on other on-line resources. (03/23/04, 28K)

Multicians updated

People who contributed to Multics, with home page and mail links. 1649 names, 594 mail addresses, 119 home pages. (254K, 06/05/04) [Add or update your entry.](#)

Features

More detail on features of Multics. (50K, 03/23/04, 3 pictures)

History

The history of Multics, from 1963 to 2000. (45K, 04/01/04, 3 images)

Stories updated

Stories and reminiscences about Multics history and its builders. (05/30/04, 28 contributors, 61 stories, 117 pictures, 4 PDFs)

Myths About Multics updated

There are a lot of incorrect statements about Multics online. This section tries to correct some of them. (48K, 06/02/04)

PL/I

How Multics was built in a high-level language. Compiler construction and compiler features. (46K, 03/23/04)

Bibliography

References to published information on Multics. (186K, 05/08/04, 527 documents, 112 online)

Technical Papers

A selection of technical papers about Multics is available on-line. (18K, 03/23/04, 26 papers)

Chronology

Chronological list of dates in the history of Multics. (32K, 03/23/04)

Site Listing updated

83 Multics sites. (22K, 05/31/04)

Site Timeline updated

Chart showing when sites were installed and uninstalled. (31K, 05/31/04)

Glossary updated

An encyclopedic, cross-referenced dictionary of Multics terms, concepts, and jargon, with contributions by many Multicians. (27 files, 05/31/04, 796 entries)

[Index](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#)

Humor

Jokes about Multics. (25K, 1 picture, 03/23/04)

Development Documents

The documents we produced while building Multics. (14K, 03/23/04)

A few samples from these hundreds of documents are on-line:

- [Multics Design Notebook, section I: Introduction](#) by Professor Corbató (13K, 11/30/64, posted 10/10/01)
- [AW17: Multics Commands and Active Functions](#) pocket guide (81K, 04/01/80, posted 05/07/99)
- [AG91: Multics Programmers' Manual: Reference Guide](#) Table of Contents (170K, 1984, posted 11/28/98)
- [Reserved Segment Name Suffixes](#) AG91 Appendix E (8K, 11/24/98)
- [BA.1 MSPM- General Information](#) (5K, 07/26/68)
- [BA.2 Summary of Multics Technical Policy](#) (5K, 07/26/65)
- [BA.3.00 Documentation Conventions](#) (8K, 05/19/66)
- [BB.2 System Module Interfaces \(PL/I Subset for System Programming\)](#) (19K, 9 figures, 6/24/66)
- [BB.2.01 EPL Subset for System Programming](#) (6K, 01/31/69)
- [BB.3.01 Multics Standard Tape Format](#) (9K, 06/02/67)
- [BX.1.00 Multics Command Language](#) (18K, 10/29/68)
- [Multics Technical Bulletins Index](#) (234K, 11/05/87, 719 documents)
- [MAB-048 Rules for the Multics Change Review Board](#) (20K, 08/12/85)

Program Source

Samples of Multics source code in PL/I and ALM. (7K, 03/23/04, 8 files)

[Index to the Multics source files](#) (596K, 05/23/01, 5839 entries).

Other Web Sites

Links to other Internet sites of interest to Multicians. (23K, 03/23/04)

Image Gallery updated

A visual index to all the graphic images on the site, linked to where they are used. (201 thumbnails -- high speed connection recommended) (45K, 06/07/04)



Site Histories

Stories and pictures about Multics sites past and present. Eventually we should have one for each site.

- [AFDSC: US Air Force Data Services Center](#), Pentagon, Washington DC (12K, 11/29/02) **updated**
- [CISL: Honeywell Cambridge Information Systems Laboratory](#), Cambridge, MA (7K, 09/20/95)
- [DOCKMASTER: US National Security Agency](#), Linthicum, MD (11K, 07/20/99)
- [FORD: Ford Motor Company](#), Dearborn, MI (18K, 03/23/04, 7 images)
- [GM: General Motors Corporation](#), Detroit, MI (16K, 10/20/98)
- [DND-H: Canadian Department of National Defence, MCHQ](#), Halifax, NS (16K, 11/05/00)
- [MIT: Massachusetts Institute of Technology](#), Cambridge, MA (13K, 07/29/02, 1 image)
- [NWGS: Naval War Games Systems](#), 4 systems (17K, 01/14/98)
- [OU: Oakland University](#), Rochester, MI (14K, 02/03/98)
- [PRHA: Puerto Rican Highway Authority](#), San Juan, PR (12K, 03/23/04, 5 pictures)
- [RADC: US Air Force Rome Air Development Center](#), Griffiss Air Force Base, Rome, NY (15K, 03/23/04)
- [STC: Standard Telephones and Cables](#), New Southgate, London, England (12K, 05/19/04)
- [System-M: Honeywell Information Systems](#), Phoenix, AZ (9K, 03/23/04)
- [Systeme X: Bull Multics Support](#), Louveciennes/Rocquencourt, France (5K, 09/22/02, 6 pictures)
- [UC: University of Calgary and ACTC/Perigon/CGI](#), Calgary, Alberta, Canada (11K, 03/23/04)
- [USGS: US Geological Survey](#), Reston, Denver, Menlo Park (9K, 03/23/04)
- [USL: University of Southwestern Louisiana](#), Lafayette, LA (8K, 03/23/04, 3 pictures)

Usenet News

The newsgroup [alt.os.multics](#) is active and discusses Multics features and history, influence on other systems, Multicians, and the usual newsgroup noise.

[Editor: thvv@multicians.org](mailto:thvv@multicians.org)



Computer Modell Katalog



CMK durchsuchen nach :
[FreeFind](#)

powered by

	Find!	Site Map

☒ CMK SUCHE ☐ WEB SUCHE

Einleitung

1. Einleitung

- 1. Einleitung
- 2. Grundlegung
- 2.1. Rechnergenerationen im Überblick
- 2.2. Betriebssystemgenerationen im Überblick
- 2.3. von Neumann-Rechnerarchitektur früher und heute
- 2.4. Leistungsvergleich früherer Rechner und heutiger PCs

Keine Branche hat sich in den letzten 50 Jahren so schnell entwickelt, wie die Computerbranche. Während heute PCs, PDAs, Handys und andere multifunktionale Geräte aus dem Alltag nicht mehr wegzudenken sind, standen Computer zu Beginn nur sehr wenigen Institutionen zur Verfügung. Computer passten nicht in die Jackentasche, sondern belegten ganze Räume – bei gleichzeitig sehr viel geringerer Rechenleistung. Gleichzeitig waren die Kosten für die Hardware immens, die Handhabung ebenfalls nicht mit der eines heutigen PCs zu vergleichen. Um die Entwicklung zu verdeutlichen, lieferte Bill Gates, Mitbegründer und langjähriger CEO des Softwarekonzerns Microsoft, einen Vergleich mit der Automobilindustrie: „Wenn General Motors (GM) mit der Technologie so mitgehalten hätte wie die Computerindustrie, dann hätten wir heute alle 25-Dollar-Autos, die 1000 Meilen pro Gallone Sprit fahren würden.“ Das jedoch die Entwicklungen nicht immer nur positiv einzuschätzen sind, zeigt auch die Antwort von General Motors auf den Vergleich von Bill Gates, in der General Motors in amüsanten Weise Schwächen des Microsoft-Betriebssystems auf die Automobilindustrie überträgt.

Natürlich hat sich durch die technische Entwicklung im Computer-Bereich auch die Arbeitsweise in Betrieben signifikant geändert. Nicht nur sind Verarbeitungsschritte in der Produktion automatisiert worden, auch im Bereich Forschung und Entwicklung, in der Verwaltung und Buchführung, im Personalwesen, im Marketing und in allen anderen betrieblichen Bereichen sind Computer heute nicht mehr wegzudenken. Die Produktivität erhöhte sich durch den Einsatz enorm. Neue Absatzwege sind durch das Internet hinzugekommen, durch integrierte Anwendungssysteme sind die betrieblichen Abteilungen zusammengedrückt. Auch hier haben sich – unter anderem auch durch neue Betriebssysteme, Standardsoftware, Netzwerke und Anwendungen – viele Veränderungen ergeben.



Chinesische Suan-pan (Abakus), 19. Jh.
Deutsches Museum, München

Diplomarbeit
Dennis Bode
06.01.2004

dbode@hardwareluxx.com



In dieser Arbeit sollen die letzten 50 Jahre Computergeschichte, die Entwicklungen im Hard- und Softwarebereich, sowie die Auswirkungen auf die betrieblichen Anwendungssysteme aufgezeigt werden. Einige „Meilensteine“ aus dieser Entwicklung, die einen besonderen Anteil an der heutigen Computer-Welt haben, sollen zudem detailliert abgebildet werden, um Zusammenhänge und Auswirkungen zu verdeutlichen. In jeder dargestellten Periode ergeben sich aufgrund der Innovationen bestimmte Errungenschaften und Fortschritte, die ebenso transparent gemacht werden sollen, wie die Mängel und Probleme der jeweiligen Technologien.

Zunächst sollen in einem Überblick grundlegende Begriffe aufgezeigt werden, welche zum Verständnis der historischen Entwicklung notwendig sind. Es soll eine Einordnung in Rechnergenerationen stattfinden, durch die sich die weitere Darstellung der Arbeit manifestiert. Darüber hinaus soll ein Vergleich zwischen der grundlegenden Rechnerarchitektur früher und heute gegeben werden, abschließend soll ein Leistungsvergleich Aufschluss über die Entwicklung in der Rechengeschwindigkeit geben.

Für jede Rechnergeneration sind im Hauptteil dieser Arbeit die technischen Entwicklungen, Neuheiten im Software-Bereich und die Einflüsse auf die betriebswirtschaftlichen Anwendungssysteme zu finden. Ferner werden anhand ausgewählter Entwicklungen die typischen Errungenschaften in der jeweiligen Periode dargestellt. Zum Ende der Ausführungen werden aktuelle und zukünftige Entwicklungen thematisiert und mögliche Probleme aufgezeigt.

--> weiter
2. Grundlegung
2.1. Rechnergenerationen im Überblick

Quellen :

o.V., Wiwi-Treff.de: General Motors kontert Bill Gates.



- Resource Centers
- Products & Services
- Solutions
- Technologies & Trends
- Support & Downloads

Research at Intel

- Research Areas
- Research Labs
- University Programs
- People
- News
- Events
- Open Source Software
- Research Library
- Feedback
- Search

Research & Development at Intel

Intel Technology Journal

- Research at Intel
- Research Areas
- Silicon
- Moore's Law



Silicon

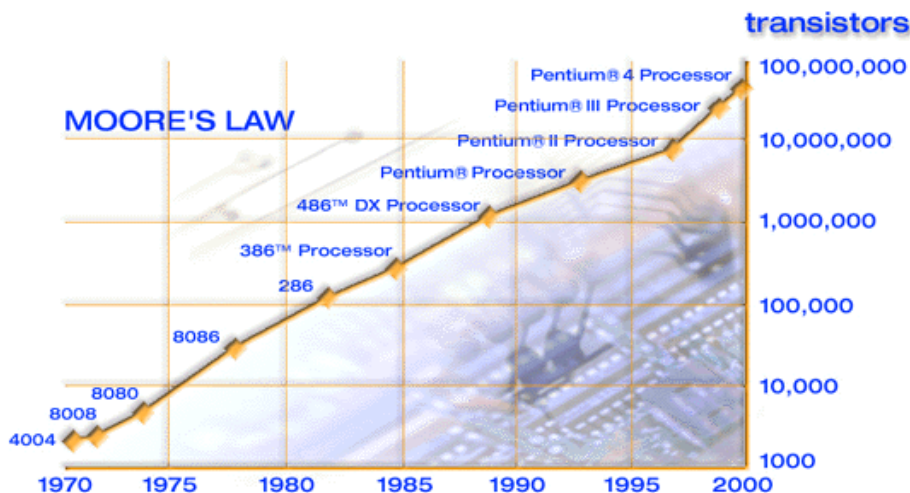
Moore's Law

Overview

Gordon Moore made his famous observation in 1965, just four years after the first planar integrated circuit was discovered. The press called it "Moore's Law" and the name has stuck. In his [original paper](#), Moore observed an exponential growth in the number of transistors per integrated circuit and predicted that this trend would continue. Through Intel's relentless technology advances, Moore's Law, the doubling of transistors every couple of years, has been maintained, and still holds true today. Intel expects that it will continue at least through the end of this decade. The mission of Intel's technology development team is to continue to break down barriers to Moore's Law.

Focus Research

- [Logic Processes](#)
- [Nanotechnology](#)
- [Lithography](#)
- [300 mm Wafers](#)
- [Flash & Other Non-Volatile Memory Technologies](#)
- [Packaging](#)



	Year of introduction	Transistors
4004	1971	2,250
8008	1972	2,500
8080	1974	5,000
8086	1978	29,000
286	1982	120,000
386™ processor	1985	275,000
486™ DX processor	1989	1,180,000
Pentium® processor	1993	3,100,000
Pentium II processor	1997	7,500,000
Pentium III processor	1999	24,000,000
Pentium 4 processor	2000	42,000,000

[back to top](#)

Gordon Moore made his famous observation in 1965, just four years after the first planar integrated circuit was discovered. The press called it "Moore's Law" and the name has stuck. In his [original paper](#), Moore observed an exponential growth in the number of transistors per integrated circuit and predicted that this trend would continue. Through Intel's relentless technology advances, Moore's Law, the doubling of transistors every couple of years, has been maintained, and still holds true today. Intel expects that it will continue at least through the end of this decade. The mission of Intel's technology development team is to continue to break down barriers to Moore's Law.



[back to top](#)

Publications

- ["Cramming More Components Onto Integrated Circuits"](#)
(Acrobat PDF file, 167 KB)
Author: Gordon E. Moore
Publication: Electronics, April 19, 1965
- ["Microprocessors Circa 2000"](#)
(Acrobat PDF file, 543 KB)
Authors: Patrick Gelsinger, Paolo Gargini, Gerhard Parker, Albert Yu
Publication: IEEE Spectrum, October 1989

Presentations

- ["No Exponential is Forever ... but We Can Delay 'Forever'"](#)
(Acrobat PDF file, 2005 KB)
Presenter: Gordon Moore
Event: [International Solid State Circuits Conference \(ISSCC\)](#) **
Date: February 10, 2003

[back to top](#)

**This link will take you off of the Intel Web site. Intel does not control the content of these linked Web sites.

Focus Research

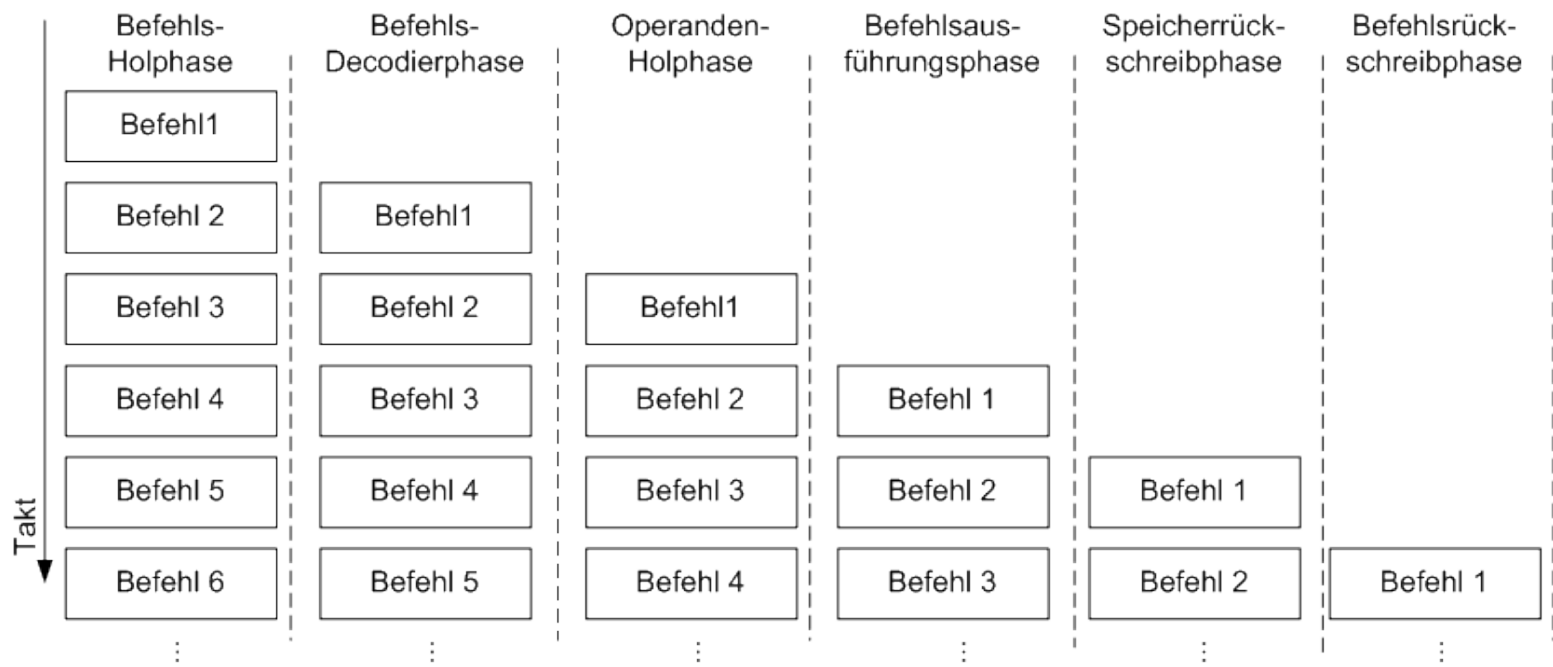
- [Logic Processes](#)
- [Nanotechnology](#)
- [Lithography](#)
- [300 mm Wafers](#)
- [Flash & Other Non-Volatile Memory Technologies](#)
- [Packaging](#)

Related Links

- [Moore's Law](#)
- [MEMS](#)
- [Lead-Free Solutions](#)
- [People](#)
- [International Technology Roadmap](#)
- [External Research Programs](#)

[Site Index](#) | [*Legal Information](#) | [Privacy Policy](#) | [Contact Us](#)

©2004 Intel Corporation





Structural Hazards

When a machine is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.

If some combination of instructions cannot be accommodated because of a **resource conflict**, the machine is said to have a structural hazard.

Common instances of structural hazards arise when

- Some functional unit is not fully pipelined. Then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle
- Some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute.

Example1:

a machine may have only one register-file write port, but in some cases the pipeline might want to perform two writes in a clock cycle.

Example2:

a machine has shared a single-memory pipeline for data and instructions. As a result, when an instruction contains a data-memory reference(load), it will conflict with the instruction reference for a later instruction (instr 3):

Clock cycle number								
Instr	1	2	3	4	5	6	7	8
Load	IF	ID	EX	MEM	WB			
Instr 1		IF	ID	EX	MEM	WB		
Instr 2			IF	ID	EX	MEM	WB	
Instr 3				IF	ID	EX	MEM	WB

To resolve this, we stall the pipeline for one clock cycle when a data-memory access occurs. The effect of the stall is actually to occupy the resources for that instruction slot. The following table shows how the stalls are actually implemented.

Clock cycle number									
Instr	1	2	3	4	5	6	7	8	9
Load	IF	ID	EX	MEM	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Stall				bubble	bubble	bubble	bubble	bubble	
Instr 3					IF	ID	EX	MEM	WB

Instruction 1 assumed not to be data-memory reference (load or store), otherwise Instruction 3 cannot start execution for the same reason as above.

To simplify the picture it is also commonly shown like this:

Clock cycle number									
Instr	1	2	3	4	5	6	7	8	9
Load	IF	ID	EX	MEM	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Instr 3				stall	IF	ID	EX	MEM	WB

Introducing stalls degrades performance as we saw [before](#). Why, then, would the designer allow structural hazards? There are two reasons:

- **To reduce cost.** For example, machines that support both an instruction and a cache access every cycle (to prevent the structural hazard of the [above example](#)) require at least twice as much total memory.
- **To reduce the latency of the unit.** The shorter latency comes from the lack of pipeline registers that introduce overhead.





Data Hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This introduces data and control hazards. **Data hazards** occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on the unpipelined machine.

Consider the pipelined execution of these instructions:

		1	2	3	4	5	6	7	8	9
ADD	R1, R2, R3	IF	ID	EX	MEM	WB				
SUB	R4, R5, R1		IF	ID _{sub}	EX	MEM	WB			
AND	R6, R1, R7			IF	ID _{and}	EX	MEM	WB		
OR	R8, R1, R9				IF	ID _{or}	EX	MEM	WB	
XOR	R10, R1, R11					IF	ID _{xor}	EX	MEM	WB

All the instructions after the ADD use the result of the ADD instruction (in R1). The ADD instruction writes the value of R1 in the WB stage (shown black), and the **SUB** instruction reads the value during ID stage (**ID_{sub}**). This problem is called **a data hazard**. Unless precautions are taken to prevent it, the SUB instruction will read the wrong value and try to use it.

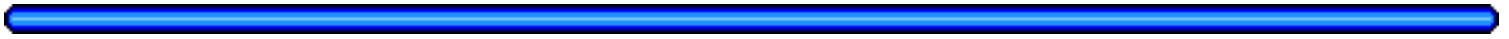
The **AND** instruction is also affected by this data hazard. The write of R1 does not complete until the end of cycle 5 (shown black). Thus, the AND instruction that reads the registers during cycle 4 (**ID_{and}**) will receive the wrong result.

The **OR** instruction can be made to operate without incurring a hazard by a simple implementation technique. **The technique** is to perform register file reads in the second half of the cycle, and writes in the first half. Because both **WB** for ADD and **ID_{or}** for OR are performed in one cycle 5, the write to register file by ADD will perform in the first half of the cycle, and the read of registers by OR will perform in the second half of the cycle.

The **XOR** instruction operates properly, because its register read occur in cycle 6 after the register write by ADD.

The next page discusses [forwarding](#), a technique to eliminate the stalls for the hazard involving the SUB and AND instructions.

We will also [classify the data hazards](#) and consider the cases [when stalls can not be eliminated](#). We will see what compiler can do to [schedule the pipeline](#) to avoid stalls.





Control Hazards

Control hazards can cause a greater performance loss for DLX pipeline than data hazards. When a branch is executed, it may or may not change the PC (program counter) to something other than its current value plus 4. If a branch changes the PC to its target address, it is a **taken branch**; if it falls through, it is **not taken**.

If instruction *i* is a **taken branch**, then the PC is normally not changed until the end of MEM stage, after the completion of the address calculation and comparison ([see diagram](#)).

The simplest method of dealing with branches is **to stall** the pipeline as soon as the branch is detected until we reach the **MEM** stage, which determines the new PC. The pipeline behavior looks like :

Branch	IF	ID	EX	MEM	WB					
Branch successor		IF(stall)	stall	stall	IF	ID	EX	MEM	WB	
Branch successor+1						IF	ID	EX	MEM	WB

The stall does not occur until after **ID** stage (where we know that the instruction is a branch).

This control hazards stall must be implemented differently from a data hazard, since the **IF** cycle of the instruction following the branch **must be repeated** as soon as we know the branch outcome. Thus, the first **IF** cycle is essentially a **stall** (because it never performs useful work), which comes to total 3 stalls.

Three clock cycles wasted for every branch is a significant loss. With a 30% branch frequency and an ideal CPI of 1, the machine with branch stalls achieves only **half** the ideal speedup from pipelining!

The number of clock cycles can be reduced by two steps:

- Find out whether the branch is taken or not taken **earlier** in the pipeline;
- Compute the taken PC (i.e., the address of the branch target) **earlier**.

Both steps should be taken as early in the pipeline as possible.

By moving the zero test into the ID stage, it is possible to know if the branch is taken at the end of the

ID cycle. Computing the branch target address during ID requires an additional adder, because the main ALU, which has been used for this function so far, is not usable until EX.

The revised datapath :

With this datapath we will need only one-clock-cycle stall on branches.

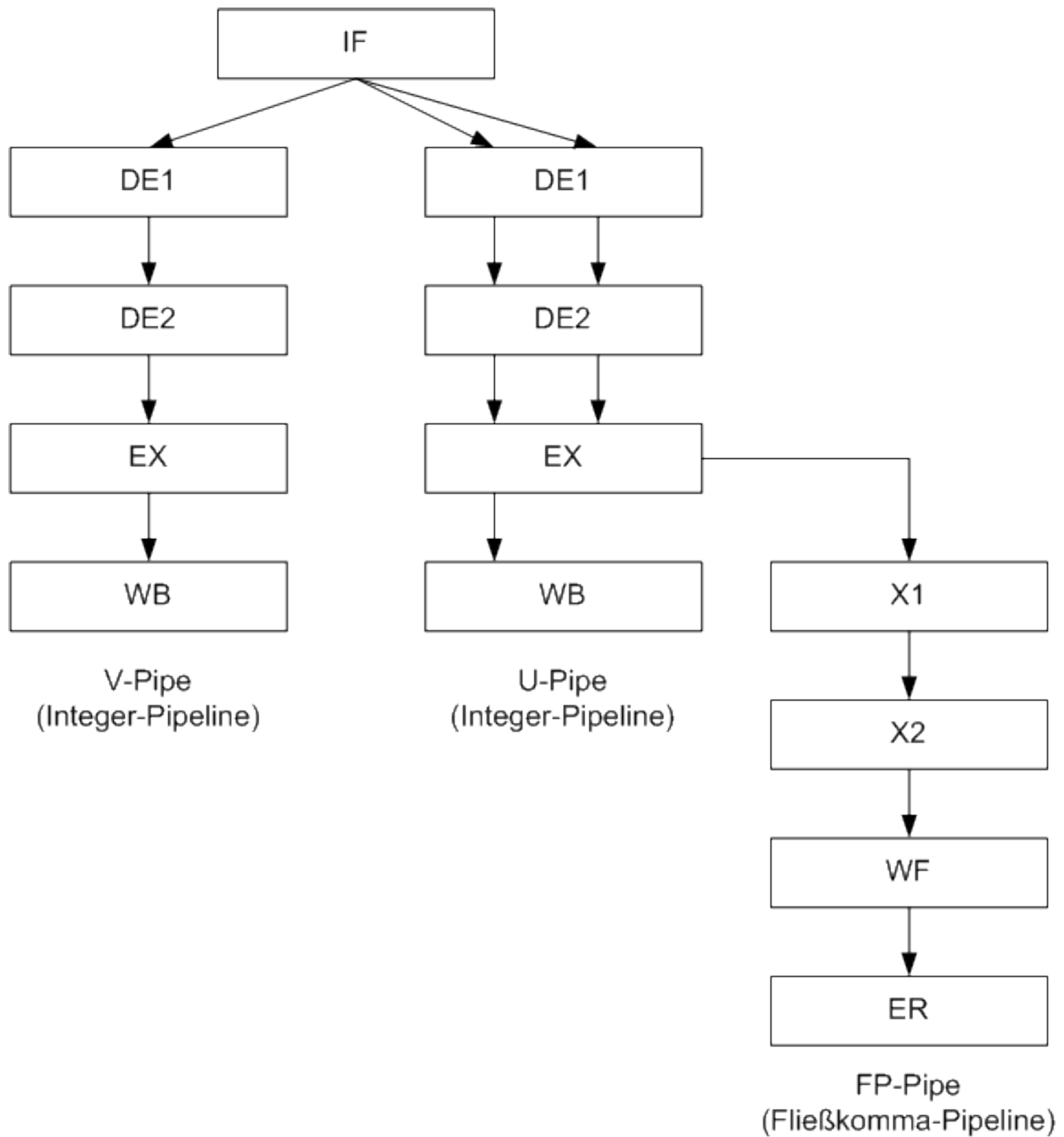
Branch	IF	ID	EX	MEM	WB		
Branch successor		IF(stall)	IF	ID	EX	MEM	WB

In some machines, branch hazards are even more expensive in clock cycles. For example, a machine with separate decode and register fetch stages will probably have a *branch delay* - the length of the control hazard - that is at least one clock cycle longer. The branch delay, unless it is dealt with, turns into a *branch penalty*. Many older machines that implement more complex instruction sets have branch delays of four clock cycles or more.

In general, the deeper the pipeline, the worse the branch penalty in clock cycles.

There are many methods for dealing with the pipeline stalls caused by branch delay. Further we will discuss four simple [branch prediction schemes](#). Then we will look at more powerful compile-time scheme such as [loop unrolling](#) that reduces the frequency of loop branches.







Pipeline Hazards

There are situations, called **hazards**, that prevent the next instruction in the instruction stream from being executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining.

There are three classes of hazards:

- Structural Hazards. They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
- Data Hazards. They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
- Control Hazards. They arise from the pipelining of branches and other instructions that change the PC.

Hazards in pipelines can make it necessary to stall the pipeline. The processor can stall on different events:

- A **cache miss**. A cache miss stalls all the instructions on pipeline both before and after the instruction causing the miss.
- A **hazard in pipeline**. Eliminating a hazard often requires that some instructions in the pipeline to be allowed to proceed while others are delayed. When the instruction is stalled, all the instructions issued *later* than the stalled instruction are also stalled. Instructions issued *earlier* than the stalled instruction must continue, since otherwise the hazard will never clear.

A hazard causes pipeline bubbles to be inserted. The following table shows how the stalls are actually implemented. As a result, no new instructions are fetched during clock cycle 4, no instruction will finish during clock cycle 8.

In case of structural hazards:

Clock cycle number										
Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	EX	MEM	WB				

Instr i+2			IF	ID	EX	MEM	WB			
Stall				bubble	bubble	bubble	bubble	bubble		
Instr i+3					IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

To simplify the picture it is also commonly shown like this:

Clock cycle number										
Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	EX	MEM	WB				
Instr i+2			IF	ID	EX	MEM	WB			
Instr i+3				stall	IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

In case of data hazards:

Clock cycle number										
Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	bubble	EX	MEM	WB			
Instr i+2			IF	bubble	ID	EX	MEM	WB		
Instr i+3				bubble	IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

which appears the same with stalls:

Clock cycle number										
Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	stall	EX	MEM	WB			
Instr i+2			IF	stall	ID	EX	MEM	WB		
Instr i+3				stall	IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

Performance of Pipelines with Stalls

A stall causes the pipeline performance to degrade the ideal performance.

Average instruction time pipelined

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{CPI}_{\text{unpipelined}} * \text{Clock Cycle Time}_{\text{unpipelined}}} \\ = \frac{\text{CPI}_{\text{unpipelined}} * \text{Clock Cycle Time}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}} * \text{Clock Cycle Time}_{\text{pipelined}}}$$

The ideal CPI on a pipelined machine is almost always 1. Hence, the pipelined CPI is

$$\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction} \\ = 1 + \text{Pipeline stall clock cycles per instruction}$$

If we ignore the cycle time overhead of pipelining and assume the stages are all perfectly balanced, then the cycle time of the two machines are equal and

$$\text{Speedup} = \frac{\text{CPI}_{\text{unpipelined}}}{1 + \text{Pipeline stall cycles per instruction}}$$

If all instructions take the same number of cycles, which must also equal the number of pipeline stages (the depth of the pipeline) then unpipelined CPI is equal to the depth of the pipeline, leading to

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls, this leads to the intuitive result that pipelining can improve performance by the depth of pipeline.




[Home](#)
[About](#)
[Forums](#)
[Mobile](#)
[News](#)
[Articles](#)
[RealTime Pricing](#)
[System Rigs](#)
[My AnandTech](#)
[Tech FAQs](#)
[Vendor Ratings](#)
[Team AnandTech](#)
[Anand's Weblog](#)

Intel's Pentium 4 E: Prescott Arrives with Luggage


Date: February 1st, 2004

Topic: CPU

Manufacturer: Intel

Author: Anand Lal Shimpi & Derek Wilson

AnandTech Deals

[Intel PENTIUM 4 3.2GHZ \(bx80546pg3200e\) Processor Products](#)

Prices

- > [NewEgg.com](#) \$ 279.00
- > [Compuplus.com](#) \$ 289.00
- > [Shopperwiz.com](#) \$ 294.32

Pipelining: 101

It seems like every time Intel releases a new processor we have to revisit the topic of pipelining to help explain why a 3GHz P4 performs like a 2GHz Athlon 64. With a 55% longer pipeline than Northwood, Prescott forces us to revisit this age old topic once again.

You've heard it countless times before: pipelining is to a CPU as the assembly line is to a car plant. A CPU's pipeline is not a physical pipe that data goes into and appears at the end of, instead it is a collection of "things to do" in order to execute instructions. Every instruction must go through the same steps, and we call these steps stages.

The stages of a pipeline do things like find out what instruction to execute next, find out what two numbers are going to be added together, find out where to store the result, perform the add, etc...

The most basic CPU pipeline can be divided into 5 stages:

1. Instruction Fetch
2. Decode Instructions
3. Fetch Operands
4. Execute
5. Store to Cache

You'll notice that those five stages are very general in their description, at the same time you could make a longer pipeline with more specific stages:

1. Instruction Fetch 1
2. Instruction Fetch 2
3. Decode 1
4. Decode 2
5. Fetch Operands
6. Dispatch
7. Schedule
8. Execute
9. Store to Cache 1
10. Store to Cache 2

Both pipelines have to accomplish the same task: instructions come in, results go out. The difference is that each of the five stages of the first pipeline must do more work than each of the ten stages of the second pipeline.

If all else were the same, you'd want a 5-stage pipeline like the first case, simply because it's easier to fill 5 stages with data than it is to fill 10. And if your pipeline is not constantly full of data, you're losing precious execution power - meaning your CPU isn't running as efficiently as it could.

The only reason you would want the second pipeline is if, by making each stage simpler, you can get the time it takes to complete each stage to be significantly quicker than in the previous design. Your slowest (most complicated) stage determines how quickly you can get data through each stage - keep that in mind.

Let's say that the first pipeline results in each stage taking 1ns to complete and if each stage takes 1 clock cycle to execute, we can build a 1GHz processor ($1/1\text{ns} = 1\text{GHz}$) using this pipeline. Now in order to make up for the fact that we have more stages (and thus have more of a difficult time keeping the pipeline full), the second design must have a significantly shorter clock period (the amount of time each stage takes to complete) in order to offer equal/greater performance to the first design. Thankfully, since we're doing less work per clock - we can reduce the clock period significantly. Assuming that we've done our design homework well, let's say we get the clock period down to 0.5ns for the second design.

Design 2 can now scale to 2GHz, twice the clock speed of the original CPU and we will get twice the performance - assuming we can keep the pipeline filled at all times. Reality sets in and it becomes clear that without some fancy footwork, we can't keep that pipeline full all the time - and all of the sudden our 2GHz CPU isn't performing twice as fast as our 1GHz part.

Search AnandTech

[advanced search](#)

CPU News

[AMD's Opteron 150 and 250 processors](#)
[Kicking It Down A Notch : AMD's Athlon64 2800+ Processor](#)
[64-bit vs 32-bit Performance \[Athlon 64\]](#)
[AMD Athlon 64 2800+ Processor Review](#)
[Intel Celeron 2.8GHz Review](#)
[AMD Athlon 64 3000+ Processor Performance](#)
[Intel Pentium 4 3.4E Reviews](#)
[AMD Athlon 64 Model 3000+ Review](#)
[more News](#)

Latest CPU Articles

[AMD Athlon 64 3800+ and FX-53: The First 939 CPUs](#)
[AMD Update: Following in Intel's Naming Footsteps](#)
[AMD Athlon 64 2800+: A Cheaper Newcastel](#)
[Tech Update: AMD core changes on the horizon](#)
[CPU Heat Comparison: How Hot is Prescott?](#)
[AT News Update: Socket 775 Processor Names](#)
[The Athlon 64 FX-53: AMD's Next Enthusiast Part](#)
[AT News Update: Centrino and Xeon 2004 Roadmaps](#)
[more CPU Articles](#)

Latest CPU forum posts

[Can't seem to overclock by more than 5mhz FSB...](#)
[xeon 2.4ghz](#)
[The Best CPU WITHOUT Needing To OC It?](#)
[Need help setting up my bios](#)
[how to overclock this beast](#)
[Cold boot problems at 270FSB on Abit IS7-G with P4 2.4C](#)
[Has it been confirmed that AMD's Paris chip will use socket 754, or is this a rumor?](#)
[Bad chip for Overclocking?](#)
[Overclocking the AMD Athlon XP2600+ Mobile Help?](#)
[eXcaliburPC CPU's](#)
[more CPU Discussions](#)
[Newsletter](#)

AT DealFinder

Enter Product:

My AnandTech

Username Password



Make sense? Now let's relate this to the topic at hand.

31 Stages: What's this, Baskin Robbins? 



Review Index:

Select



Print this article



Find the lowest prices



Email this article

> Buy it from NewEgg.com for \$ 279.00



104 comment(s) - last comment Jeff7181 on Mar 11, 2004 at 3:27 AM

Sponsored Links (Get Listed)

[Better virus protection starts inside.](#) Resellers! Help protect your customers from certain viruses with the AMD Athlon(TM) 64 processor with Enhanced Virus Protection for the upcoming Microsoft (R) Windows(R) XP SP2.

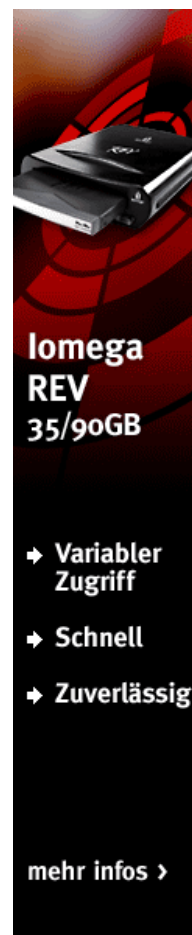
[Get the best computer deals only @ TigerDirect.com](#) We have what you are looking for in stock & ready to ship at a great price! Computers, Notebooks, Monitors, Digital Cameras, CPU's, Memory, Motherboards, Barebone kits, Networking, Video, & more at the best values. Order today, ships today!

[Laptop CPUs, Laptop Accessories, Hard Drives](#) Shop for laptop parts & accessories for all major brands, from Dell, Sony, Toshiba, Compaq, IBM, Gateway & more. Save on batteries, hard drives, power adapters, memory upgrades, modems, keyboards Receive same day shipping. Yahoo Secured site.

[Powerful ProStar & Sager Laptops](#) Wholesale prices with free shipping. Easy Returns and live sales chat.

[Velocity Micro Gaming Computers](#) Performance matters. Velocity Micro proves a computer is more than the sum of it's components. Experience our powerful Velocity Generator configuration tools to let us build your dream system!

Copyright © 1997-2003 AnandTech, Inc. All rights reserved. Terms, Conditions and privacy information. Site Design by Anders Hammervald
[Click Here for Advertising Information](#)



Search Prices

Shop a Category
Shopping Keywords
 Shop
Powered by BizRate

Processors



Hot!

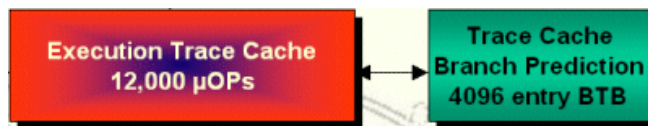
Computex 2004: Day 1



Despite reports of a new SARS outbreak in mainland China, the mood at Computex 2004 remains upbeat with plenty of goodies to see and touch. AMD launches its Socket 939 architecture, new mobos from Abit, Ali, Asus and VIA, more cases from Acer and more.

Intel's New Pentium 4 Processor

The Trace Cache Branch Prediction Unit



Intel is very proud on the branch prediction unit that aids the execution trace cache. Its branch target buffer is 8 times as large as the one found in Pentium III and its new algorithm is supposed to be way better than AMD's latest G-share algorithm used in Thunderbird and Spitfire. Intel claims that this unit can eliminate 33% of the mispredictions of Pentium III.

Search Articles

Select a Category
Search Keywords
 Search

Hyper Pipeline

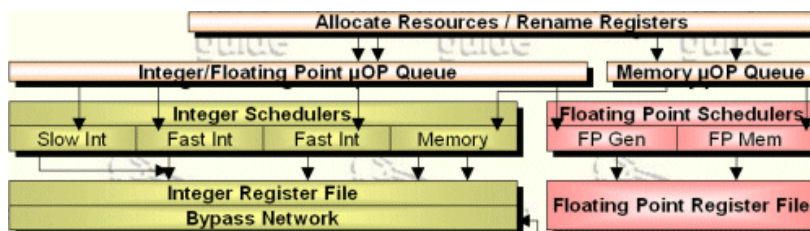


One of the most well known features of the new Pentium 4 is its extremely long pipeline. While the pipeline of Pentium III has 10 stages and the one of Athlon 11, Pentium 4 has no less than 20 stages.

The reason for the longer pipeline is Intel's wish of Pentium 4 to deliver highest clock rates. The smaller or shorter each pipeline stage, the fewer transistors or 'gates' it needs and the faster it is able to run. However, there is also one big disadvantage to long pipelines. As soon as it turns out at the end of the pipeline that the software will branch to an address that was not predicted, the whole pipeline needs to be flushed and refilled. The longer the pipeline the more 'in-flight' instructions will be lost and the longer it takes until the pipeline is filled again.

Intel is proud to announce that the Pentium 4 pipeline can keep up to 126 instructions 'in-flight', amongst them up to 48 load and 24 store operations. The improved trace cache branch prediction unit described above is supposed to ensure that flushes of this long pipeline are only rare occasions.

The stuff that happens in the trace cache, as mentioned above, only represents the first five stages of the pipeline of Pentium 4. What follows is



- Allocate resources
- Register renaming
- Write into the μ OP queue
- Write into the schedulers and compute dependencies
- Dispatch μ OPs to their execution units
- Read register file (to ensure that the correct ones of the 128 all-purpose register files are used as the register(s) for the actual instruction)

After that comes the actual execution of the μ OP, which I will discuss more detailed in the next paragraph. Of the above-mentioned previous stages the schedulers as well as the register file read are the most interesting. I have still decided against discussing them in detail to keep this article from becoming my next book.

[BACK](#) | [NEXT: The Rapid Execution Engine](#)

Search for lowest prices on top products

Search in: Computer Hardware Go!

[Home](#)

[Latest Hard News](#)

[Hard Newsletter](#)

[Tom's Guides](#)

[Community](#)

[About Us](#)

[Service](#)

[Search Prices](#)

[Search Articles](#)

[Keyword Index](#)

[Historical Archive](#)

[LAN Parties](#)

[THG Videos](#)

[Product Database](#)

[Information](#)

[Advertising Info](#)

[Privacy Policy](#)

Sponsored Link

[Better virus protection starts inside.](#)

Resellers! Help protect your customers from certain viruses with the AMD Athlon(TM) 64 processor with Enhanced Virus Protection for the upcoming Microsoft(R) Windows(R) XP SP2.

Related Links

[HOT! Update Of Intel Roadmap News!](#)

[Intel Roadmap News 10/2000 - Part Two, Intel's Future Mobile and Server / Workstation Products](#)

[Intel Roadmap News 10/2000 - Part One, Desktop Processors And Chipsets](#)

[AMD vs. Intel: The best CPU for MPEG-4.](#)

[DDR-SDRAM Has Finally Arrived](#)

Find the lowest prices on:

digital cameras: canon, sony, olympus, nikon, fuji, minolta

pdas: palm, sony, handspring, hewlett packard, compaq

graphics cards: ati, radeon, all in wonder, nvidia, geforce fx

monitors: flat panel, viewsonic, nec, samsung, philips, benq

cases: chenbro, cooler master, thermaltake, chieftec

mainboards: asus, shuttle, abit, gigabyte, biostar, msi, aopen

cpus: amd, athlon, intel, celeron, opteron

ram: corsair, kingston, samsung, crucial, ocz

storage: ibm, western digital, firewire, seagate, archos, cms

tom's low price finder

Modding The Xbox Into The Ultimate Multimedia Center



Like a teenage kid with a genius IQ who is addicted to video games, Microsoft's Xbox offers a lot of untapped potential. THG downloads a version of Xbox Media Center software and installs it on an Xbox mod from FriendTech to show just what the game console is capable of.

VGA Charts Part III

This year's THG VGA charts feature performance tests from ten different games and benchmarks, including an overview of 46 different graphics cards from ATI, NVIDIA, Matrox, S3 and XGI. We also included tests gauging anisotropic filtering and FSAA and a ranking of the cards by a price/quality ratio. As always, some of the results should surprise.

65 CPUs from 100 MHz to 3066 MHz

Table of Contents

After 300 hours of testing in the Tom's Hardware labs, we bring you a comparison of 65 CPUs - from the Pentium 100 to the latest Athlon XP 3000+. It's a monster review with 13 different CPU platforms.

Generation Change: Eight Motherboards For The Athlon 64

AMD's and Intel's End-of-Year CPU Buyer's Guide

NVIDIA Puts Its (New) Cards on the Table

LCD Roundup V: 17" LCDs Panels Compared

High-End PC Speakers Reviewed: It Just Doesn't Get Any More Complete Than This

Affordable and Reliable 80 and 160 GB Hard Drives

ProblemSolver: One Internet Connection - Two Private LANs

A New Notebook Hosts the Athlon64's Mobile Coming Out Party

Pinnacle ShowCenter: Symbiosis Between PC And Hi-Fi Device

THG Conference Review: Making Hardware Pay in 2004

2003 Winter Case Review Part 1: MicroATX Case Madness

Feedback

Layout!

Tell us what you think about the layout. You can also send your comments and criticism to [comment](#).

Hard Newsletter

Sign up for breaking news, reviews, and first looks in **Tom's Hard Newsletter**.

[Buy a link now](#)

Tom's Guides

Motherboards & RAM

THG Puts 13 Bleeding-Edge Memory Modules, 14 Mobos To the Match-Up Test

Graphics Cards

ATI's Optimized Texture Filtering Called Into Question

Audio Video

Altec Lansing's CS21 and Klipsch's ProMedia GMX A-2.1 Sound Better and Cost Less

Mobile Devices

Intel's Dothan Makes Its Late Debut

PCs & HowTo

THG's 15-Case Power Tower Round-Up

Games & Entertainment

Post-E3 Gaming Explosion

Processors

AMD's Socket 939 Offers More with Much of the Same

Displays

High-End LCD: Samsung and BenQ

Mass Storage

Iomega's REV Marks Leap Forward For External Drives

Peripherals & Consumer Electronics

Review: Canon Pro 1 - Sony F828 Showdown

Networking

Computex 2004 Report: The Networking View

Business Reports

Computex 2004: Day 3

AMD Extends Performance Lead With New Athlon and Duron Processor

Intel i820 Chipset Review

Intel's New Weapon - The Coppermine

Tom's Blurp - All Owners of Systems With Intel's i820 Chipset That Don't Use RDRAM Yet Will Now Get It For Free From Intel!

Tom's Blurp: Why We Don't Trust Rambus - Pointing Out Facts, Turning Rumors Into Reality

Intel Admits Problems With Pentium III 1.13 GHz - Production and Shipments Halted

Important Pentium 4 Evaluation Update

Hosted by



Sponsored Links

► [Better virus protection starts inside.](#)

Resellers! Help protect your customers from certain viruses with the AMD Athlon(TM) 64 processor with Enhanced Virus Protection for the upcoming Microsoft(R) Windows(R) XP SP2.

► [Get the best computer deals only @ TigerDirect.com](#)

We have what you are looking for in stock & ready to ship at a great price! Computers, Notebooks, Monitors, Digital Cameras, CPU's, Memory, Motherboards, Barebone kits, Networking, Video, & more at the best values. Order today, ships today!

► [Laptop CPUs, Laptop Accessories, Hard Drives](#)

Shop for laptop parts & accessories for all major brands, from Dell, Sony, Toshiba, Compaq, IBM, Gateway & more. Save on batteries, hard drives, power adapters, memory upgrades, modems, keyboards Receive same day shipping. Yahoo Secured site.

► [Hard-to-Find Computer Parts Online](#)

StarTech.com is the professionals' source for hard-to-find computer parts. Try our online Parts Finder tool to find the product you need quickly.

► [Velocity Micro Custom Computers](#)

Performance matters. Velocity Micro proves a computer is more than the sum of it's components. Experience our powerful Velocity Generator configuration tools to let us build your dream system!

Copyright of all documents and scripts belonging to this site by Tom's Guides Publishing LLC 1996 - 2004. Most of the information contained on this site is copyrighted material. It is illegal to copy or redistribute this information in any way without the expressed written consent of Tom's Guides Publishing. Please use the **[Content Permission Form](#)** for such requests. This site is NOT responsible for any damage that the information on this site may cause to your system.



Jun 2004

Serving the PC
enthusiast for over
 5×10^{-2} centuries



[Subscribe](#) to Ars Technica!

Have news? [Send it in.](#)

Ars Guides

[Buyer's Guide](#)
[How-To's & Tweaks](#)
[Product Reviews](#)
[Ars Shopping Engine](#)

Technopaedia

[Technical Blackpapers](#)
[CPU Theory & Praxis](#)
[Ars OpenForum](#)
[Search Ars](#)

Columnar Edifice

[Wankerdesk](#)
[AskArs!](#)
[Diary of a Geek](#)
[Game.Ars Report](#) [Mac.](#)
[Ars takes on...](#)
[Linux.Ars](#)

Site info

[Subscribe to Ars](#)
[Ars Merchandise](#)
[Who We Ars](#)
[Advertising](#)
[Links](#)



The Pentium 4 and the G4e: an Architectural Comparison

by [Jon "Hannibal" Stokes](#)

The perils of deep pipelining

The P4's way of doing things has advantages for certain types of applications--especially 3D and streaming media applications--but it also carries with it serious risks. The "narrow and deep" scheme gets into trouble when the processor can't find enough instructions or data in the L1 cache to keep those deep pipelines fed. Notice those empty execution slots in Figure 2.2. (The empty execution slots are the white squares, while the ones containing instructions are red). Those empty slots, called "pipeline bubbles," occur when the processor can't schedule an instruction to fit in that slot. Pipeline bubbles must propagate through each stage all the way to down the end of the pipeline, and each stage that contains a bubble with no instruction is sitting idle and wasting precious execution resources on that clock cycle.

The P4's long pipeline means that bubbles take a long time to propagate off the CPU, so a single bubble results in a *lot* of wasted

Recent Articles

[USB 2.0 Hi-Speed
Flash drive review](#)

[A closer look at
Intel's processor
numbers and 2004
road map](#)

[Far Cry game
review](#)

[Dell Latitude D800
laptop review](#)

[HP Compaq nc6000
laptop review](#)

[Hitman: Contracts
game review](#)

[Deploying a small
business Windows
2003 network](#)

[Alternative AIM
clients for Windows](#)

[Inside GNOME 2.6](#)

[Dell Latitude D600
laptop review](#)

[OWC On-The-Go](#)

cycles. When the G4e's shorter pipeline has a bubble, it propagates through to the other end quickly so that fewer cycles are wasted. So a single bubble in the P4's 20 stage pipeline wastes at least 20 clock cycles (more if it's a bubble in one of the longer FPU pipelines), whereas a single bubble in the G4e's 7 stage pipeline wastes at least 7 clock cycles. 20 clock cycles is a lot of wasted work, and even if the P4's clock is running twice as fast as the G4e's it still takes a larger performance hit for each pipeline bubble than the shorter machine.



Figure 3.1

As you can see, the more deeply pipelined a machine is, the more severe a problem pipeline bubbles become. When the P4's designers set high clock speeds as their primary goal in crafting the new microarchitecture, they had to do a *lot* of work to prevent bubbles from entering the pipeline and killing performance. In fact, we've already discussed the extremely deep scheduling queues in the P4's front end. These queues represent a place where the P4 spends a large number of transistors to alleviate the negative the effects of its long pipeline, transistors that the G4e spends instead on added execution units.

Overview of the G4e's architecture and pipeline

Mercury Pro 2.5" Firewire/USB 2.0 hard drive

Ximeta NetDisk NDU-160 review

A Look at Centrino's Core: The Pentium M

Portable headphone roundup

Deep inside KDE 3.2

OmniWeb 5.0 Beta

/etc

OpenForum

Distributed Computing

Take the Poll Technica

FAQ: Celeron overclocking

This large diagram shows the basics of the G4e's microarchitecture, with an emphasis on representing the pipeline stages of the front end and functional units. You might want to open it in a separate window and refer to it throughout this section.

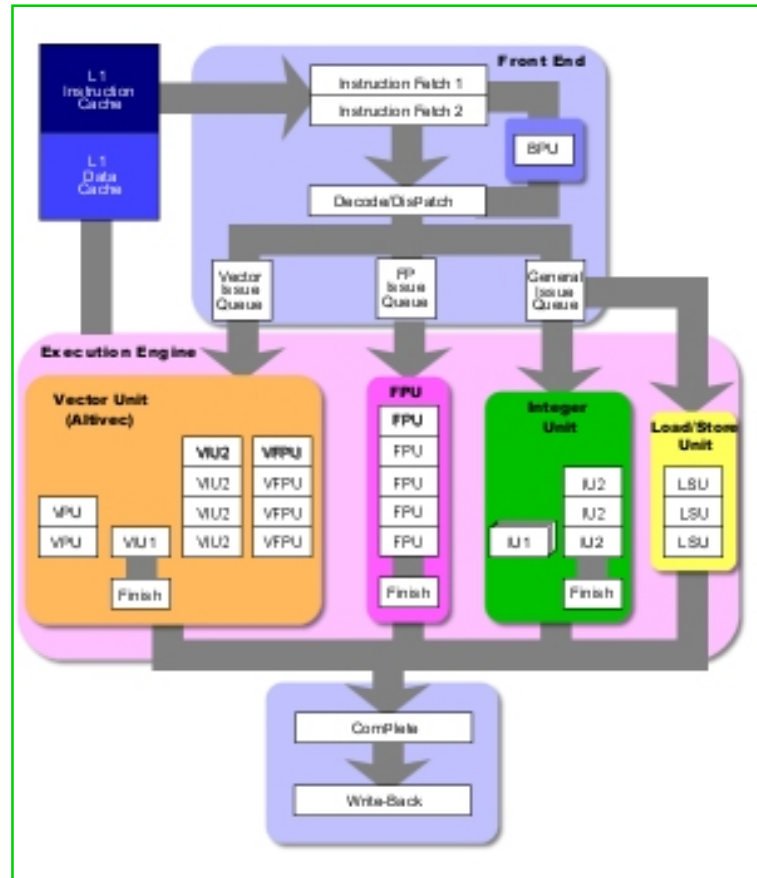


Figure 4.1: Basic Architecture of the G4e

Before instructions can enter the G4e's pipeline, they must be available in its 32K instruction cache. This instruction cache together with the 32K data cache makes up the G4e's 64K L1 cache. The instruction leaves the L1 and goes down through the various front end stages until it hits the execution engine, at which point it is executed by one of the G4e's eight execution units (not counting the Branch Processing Unit, which we'll talk about in a second).

As I've already noted, the G4e breaks down the G4's classic, four stage pipeline into seven, shorter stages:

	G4	G4e

Front End	1	Fetch	1	Fetch1
			2	Fetch2
	2	Decode/ Dispatch	3	Decode/ Dispatch
			4	Issue
Back End	3	Execute	5	Execute
	4	Complete/ Write-Back	6	Complete
			7	Write-Back

Let's take a quick look at the basic pipeline stages of the G4e, because in spite of the fact that there are more than four of them they're still pretty straightforward. An understanding of the G4e's more classic RISC pipeline will provide us with a good foundation for our upcoming discussion of the P4's much longer, more peculiar pipeline.

Stages 1 and 2 - Instruction Fetch: These two stages are both dedicated primarily to grabbing an instruction from the L1 cache. The G4e can fetch four instructions per clock cycle from the L1 cache and send them on to the next stage. Hopefully, the needed instructions are in the L1 cache. If they aren't in the L1 cache, then the G4e has to hit the much slower L2 cache to find them, which can add up to 9 cycles of delay into the instruction pipeline.

Stage 3 - Decode / Dispatch: Once an instruction has been fetched, it goes into a 12-entry instruction queue to be decoded. The decoding stage is where the processor determines what kind of instruction it's looking at and where the instruction should be sent for execution. Once the instruction is decoded, it is dispatched to the proper issue queue.

The G4e's decoder can dispatch up to three instructions per clock cycle to the next stage.

Stage 4 - Issue: There are three issue

queues for the three main types of instructions that the G4e can execute. The first queue is the Floating-Point Issue Queue (FIQ), which holds floating-point (FP) instructions that are waiting to be executed. The second queue is the Vector Issue Queue (VIQ), which holds vector (or AltiVec) operations, and the third queue is the General Instruction Queue (GIQ), which holds everything else. Once the instruction leaves its issue queue, it goes to the execution engine to be executed.

The issue stage, which also involves the reservation stations* attached to the various execution units, is the stage where the linear instruction stream is broken up into independent chunks and the instructions are rearranged and scheduled to fit the available execution resources. This is where the "out-of-order" part in "out-of-order execution" comes in. Most of this OOO action happens in the reservation stations, which do the dirty work of figuring out when to schedule instructions for the execution unit that they're attached to. The Issue Queues, however, even though they're FIFO (First In First Out) can also send instructions to the reservation stations out of program order with respect to each other.

The 4-entry Vector Issue Queue can accept up to two instructions per cycle from the dispatch unit. The 6-entry General Issue Queue can accept up to three instructions per cycle, and the 2-entry FP Issue Queue can accept one instruction per cycle.

Stage 5 - Execute: This stage is pretty straightforward. Here, the instructions can pass out of order from their issue queues into their respective functional units and be executed. Floating-point ops move into the FPU, vector ops move into one of the four AltiVec units, integer ops move into an ALU, and LOADs or STOREs move into the LOAD/

STORE Unit (LSU). We'll talk about these units in a bit more detail when we discuss the G4e's execution engine.

Stages 6 and 7 - Complete and Write-Back: In these two stages, the instructions are put back into program order (the order in which they came into the processor), and their results are written back to memory. It's important that the instructions are rearranged to reflect their original ordering so that the illusion of in-order execution is maintained to the world outside the processor. The program needs to think that its commands were executed one after the other, the way it was written.

For a look at two instructions as they travel through the G4e, check out [this animated GIF](#). Modem users should beware, though, because the GIF weights in at 355K.

Before we go into detail about the back end of the G4e, let's look at one aspect of the G4e's front end in a bit more detail: the branch processing unit.

*Note that the reservation stations are not visible in the large diagram linked above. If I were to put them in, they'd be little white boxes that sit right on top of the each group of functional units.

Next: Branch processing and prediction

Contents

- [Introduction](#)
- [Basic instruction flow](#)
- [The general approaches and design philosophies of the P4 and G4e](#)
- [The perils of deep pipelining](#)
- [Overview of the G4e's architecture and pipeline](#)
- [The branch processing unit and branch prediction](#)
- [Overview of the P4's architecture I: the trace cache](#)
- [Overview of the P4's architecture II: the pipeline](#)
- [Conclusion to Part I](#)
- [Bibliography](#)
- [Revision History](#)

Computer Architecture

The Anatomy of Modern Processors

Pipelines

In order to increase the instruction throughput, high performance processors make extensive use of a technique called **pipelining**. A pipelined processor doesn't wait until the result from a previous operation has been written back into the register files or main memory - it fetches and starts to execute the next instruction as soon as it has fetched the first one and despatched it to the instruction register. When the simple processor described on the previous pages is performing an add instruction, there is no need for it to wait for the add operation to complete before it starts fetching the next instruction. So a pipelined processor will start fetching the next instruction from memory as soon as it has latched the current instruction in the instruction register.

Thus a **pipelined processor** has a pipeline containing a number of stages (4 or 5 was a common number in early RISC processors) arranged so that a new instruction is latched into its input register as the results calculated in this stage are latched into the input register of the following stage. This means that there will be a number of instructions (equal to the number of pipeline stages in the best case) "active" in the processor at any one time.

In a typical early RISC processor (eg MIPS R3000), there would be four pipeline stages

- IF** **Instruction Fetch**
Fetch the instruction from memory
- DEC** **Decode and Operand Fetch**
Decode it and fetch operands from the register file
- EX** **Execute**
Execute the instruction in the ALU
- WB** **WriteBack**
Write the result back in to a register

<i>Time</i>	IF	DEC	EX	WB
1	i_1			
2	i_2	i_1		
3	i_3	i_2	i_1	
4	i_4	i_3	i_2	i_1
5	i_5	i_4	i_3	i_2

Population of the pipeline at each clock cycle:
 $i_1, i_2, \dots$ are successive instructions in the instruction stream.

With an n -stage pipeline, after $n-1$ clock cycles, the pipeline will become full and an instruction completes on every clock cycle, instead of every n clock cycles. This effectively speeds up the processor by a factor of n .

Latency and throughput

There's no free lunch!

Each instruction is still taking n cycles to complete. However, the **throughput** has been increased to an instruction *every* cycle from one every n cycles. The **latency** - or lag between the time an instruction is given to the processor and the time that it "emerges" with a result - has not changed. We have increased the throughput by adding a degree of parallelism to the system.

Bubbles

<i>Time</i>	IF	DEC	EX	WB
1	i_1			
2	i_2	i_1		
3	i_3	i_2	i_1	
4	i_4	i_3	i_2	i_1
5	i_4	i_3	i_2	○
6	i_4	i_3	i_2	○
7	i_5	i_4	i_3	i_2

Bubble

We can achieve the high throughput only if every instruction takes exactly one cycle to execute. Although *most* arithmetic instructions in a RISC machine complete in a single cycle, some complex instructions (eg divide) take more than a cycle. Such an instruction remains in one of the stages (eg EX) for more than a cycle and creates a **pipeline bubble** ahead of it in the pipeline.

i_2 is a long-latency instruction, eg divide, taking 3 cycles in the EX stage. A "bubble" develops ahead of it in the pipeline as instructions ahead of it move on the the next stage.

Pipeline bubbles are caused by

- Long latency instructions, such as division, floating point operations and math functions, which take more than one cycle in the execution stage,
- Memory reads, which take many cycles in the Decode stage,
- Memory writes, which may take extra cycles in the WriteBack stage, and
- Branches.

Branching in pipelined processors

Efficient handling of branch instructions presents a significant challenge to a computer architect. The simplest branch - an **unconditional** one, such as the return from a procedure or function, requires a number of instructions *behind* it in the pipeline to be *squashed*.

When i_2 the execution stage and the new PC is being calculated, instructions i_3 and i_4 have already entered the pipeline and must be squashed. This creates a series of bubbles in the pipeline until the target of the branch instruction, i_a and its successors can be fetched from memory.

The diagram shows the cost of branches fairly dramatically - a large number of bubbles are present for quite a few cycles. This diagram also makes an optimistic assumption - that the new instruction stream could be accessed within 3 cycles (not realistic if the new instructions have to come from memory rather than cache!). Branches occur very frequently in typical programs, of the order of one every 10 instructions! This shows the importance of [branch prediction strategies](#) which we will examine later.

Time	IF	DEC	EX	WB
1	i_1			
2	i_2	i_1		
3	i_3	i_2	i_1	
4	~~i_4~~	~~i_3~~	i_2	i_1
5	○	○	○	i_2
6	i_a	○	○	○
7	i_b	i_a	○	○

Branch delay slots

Some early processors were not able to squash the instruction following a branch in the hardware and required the compiler to insert a **NOOP** - an instruction that does nothing into the program following every branch.

Thus instead of emitting this:

```

mult $4, $2, $1
add $3, $4, $5
ret
sub $4, $6, $7

```

the compiler would emit:

```

mult $4, $2, $1
add $3, $4, $5
ret
or $1, $1, $1
sub $4, $6, $7

```

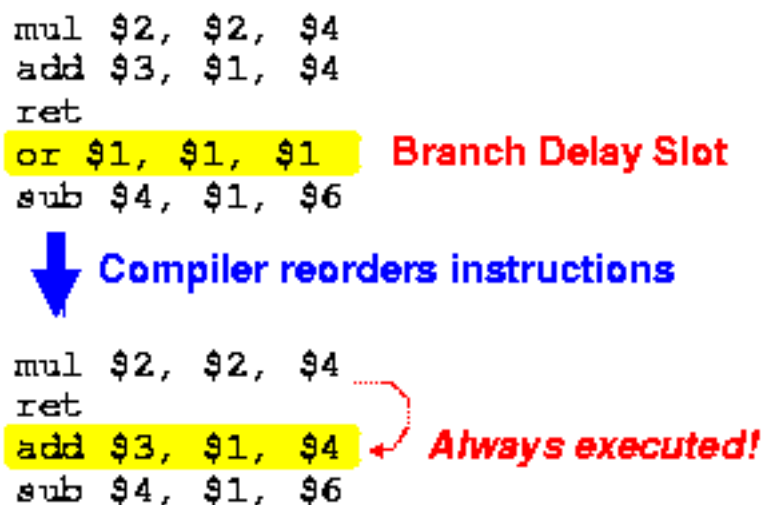
Note that
or \$1, \$1, \$1
is an *effective* NOOP - it changes nothing!

The instruction following the branch is said to be in the **branch delay slot**. It was soon realised that, since the instruction in this slot has progressed well down the pipeline anyway, if it was **guaranteed** that it would be executed, some improvement in performance would result.

The compiler is asked to move an instruction that must be executed which precedes the branch into the branch delay slot where it will be executed while the branch target is being fetched. Current RISC machines will have a one-instruction branch delay slot - occasionally two.

This is an example of the modern trend in computer architecture - to expose more details of the underlying machine to the compiler and let it generate the most efficient code. In this case, it is almost trivial for the compiler to find an instruction before the branch which *must* be executed. Compilers are rapidly becoming more capable and much more complex instruction reordering operations are routinely performed by optimising compilers.

We will see more examples of this movement of responsibility for execution efficiency from the hardware to the software later.



Conditional Branches

Branches which depend on some state of the machine (*eg* the value stored in a register) introduce another problem into pipelined machines. Consider this sequence of instructions:

```
start: addi $3, $3, 4    ; increment address
      lw $2, 0($3)      ; load next array element
      add $1, $1, $2     ; add to total
      subu $5, $5, 1     ; decrement count
      bne start          ; if not zero, branch back
```

which sums the values in an array. The element count is stored in register 5 and is decremented on each iteration of the loop. The loop terminates when it reaches 0. The branch tests the condition code generated by the `dec` operation. If it's not zero, it will branch back to the beginning of the loop. However this value is not available until the `dec` instruction has reached the **WB** stage. The branch instruction must therefore **stall** in the execution stage because one of its operands (the condition code) was not available when it entered this stage. This is the first example of a **pipeline hazard** - we shall see more in the [next section](#).

Additional References

- [Pipelines from CS380, BYU](#)
- [Pipeline Lectures from CSE 322 at Notre Dame \(.ps\)](#)

Continue on to [Pipeline Hazards](#)

Back to the [Table of Contents](#)

© [John Morris](#), 1998

Pipeline Streaming

Abstract

The purpose of this document is to provide an overview of pipelining computer processors. The topic will be covered in general with a focus on some special topics of interest. One area of focus is some of the early criticisms of pipelining and why, in retrospect, they were wrong. The emergence and evolution of pipelining in the early IBM line of mainframe computers, including the IBM 7030, better known as the "Stretch" computer will be discussed. This will be contrasted with the state of pipelining in the current generation of microprocessors from Intel and Motorola.

Introduction

A Pipeline is a series of stages, where some work is done at each stage. The work is not finished until it has passed through all stages. Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, Pipelining is key to making processors fast. A pipeline is like an assembly line: in both, each step completes one piece of the whole job. Workers on a car assembly line perform small tasks, such as installing seat covers. The power of the assembly line comes from many cars per day. On a well-balanced assembly line, a new car exits the line in the time it takes to perform one of the many steps. Note that the assembly line does not reduce the time it takes to complete an individual car; it increases the number of cars being built simultaneously and thus the rate at which the cars are started and completed. There are two types of pipelines, Instructional pipeline where different stages of an instruction fetch and execution are handled in a pipeline and Arithmetic pipeline where different stages of an arithmetic operation are handled along the stages of a pipeline.

Pipelining is used to obtain improvements in processing time that would be unobtainable with existing non-pipelined technology. The development goal for the IBM 7030 (the Stretch Computer) was an over-all performance of 100 times the 704 computer, the fastest computer in production at that time, whereas circuit improvements would only give a factor-of-10 improvement [CAM62, p 1-2]. This goal could only be met with overlapping instructions, i.e. pipelining.

A Pipeline is used to improve performance beyond what can be achieved with non-pipelined

processing. Similarly, the goal for the IBM 360/91 was an improvement of one to two orders of magnitude over the 7090. Technology advances could only bring about a four-fold improvement [AND67, p 8].

In a more recent example, the 6502 microprocessor had a through-put similar to the 8080 processor running at a clock rate four times faster. This was due to the pipelined architecture of the 6502 versus the non-pipelined 8080 [HEN94].

There are two disadvantages of pipeline architecture. The first is complexity. The second is the inability to continuously run the pipeline at full speed, i.e. the pipeline stalls. There are many reasons as to why pipeline cannot run at full speed. There are phenomena called pipeline hazards, which disrupt the smooth execution of the pipeline. The resulting delays in the pipeline flow are called bubbles. These pipeline hazards include

- structural hazards from hardware conflicts
- data hazards arising from data dependencies
- control hazards that come about from branch, jump, and other control flow changes

These issues can and are successfully dealt with. But detecting and avoiding the hazards leads to a considerable increase in hardware complexity. The control paths controlling the gating between stages can contain more circuit levels than the data paths being controlled [AND67, p 9]. In 1970, this complexity is one reason that led Foster to call pipelining still controversial [FOS76, p253-256].

The one major idea that is still controversial is "instruction look-ahead" [pipelining]...

Why then the controversy? First, there is a considerable increase in hardware complexity. The second problem when a branch instruction comes along, it is impossible to know in advance of execution which path the program is going to take and, if the machine guesses wrong, all the partially processed instructions in the pipeline are useless and must be replaced. In the second edition of Foster's book, published 1976, this passage was gone. Apparently, Foster felt that pipelining was no longer controversial.

Doran also alludes to the nature of the problem. The model of pipelining is "amazingly simple" while the implementation is "very complex" and has many complications [DOR79, p 217].

Because of the multiple instructions that can be in various stages of execution at any given moment in time, handling an interrupt is one of the more complex tasks. In the IBM 360, this can lead to several instructions executing after the interrupt is signaled, resulting in an imprecise interrupt [AND67, p 14]. An imprecise interrupt can result from an instruction exception and precise address of the instruction causing the exception may not be known! This led Myers to criticize pipelining, referring to the imprecise interrupt as an "architectural nuisance". He stated that it was not an advance in computer architecture but an improvement in implementation that could be viewed as a step backward [MYE78, p 9].

In retrospect, most of Myers' book *Advances in Computer Architecture* dealt with his concepts for improvements in computer architecture that would be termed CISC today. With the benefits of hindsight, we can see that pipelining is here today and that most of the new CPUs are in the RISC class. In fact, Myers is one of the co-architects of Intel's series of 32-bit RISC microprocessors. This processor is fully pipelined [MYE88]. I suspect that Myers no longer considers pipelining a step backwards.

The difficulty arising from imprecise interrupts should be viewed as a complexity to be overcome, not as an inherent flaw in pipelining. Doran explains how the B7700 carries the address of the instruction through the pipeline, so that any exception that the instruction may raise can be precisely located and not generate an imprecise interrupt [DOR79, p 219].

Earliest development:

Overlapping instructions in IBM 704/709/7094 computers

The IBM 704 computer was a vacuum tube computer that used core memory. It had no overlap in its instruction execution with input/output. The 709 was basically a 704 with some architectural addition. The most important of these was data-synchronizer units (now called channels) which allowed instruction execution and input/output to overlap. This allowed the 709 -- with the same clock cycle as the 704 -- to run about 60% faster. [STA87].

The IBM 7094 I was a transistor-based machine that still used core memory. This cpu includes an

Instruction Backup Register (IBR) which was used to buffer the next instruction. This was used to overlap the execution of one instruction with the fetch of the next. The cycle time was about 6 times faster than the 709 but the system ran about 7.5 times faster than the 709. Hence, this rudimentary pipelining achieved a performance increase in about 25%. While neither of these gains were tremendous, they pointed the way to continued improvement in performance beyond the increase in circuit speed.

Pipelining: History

The IBM 360 Series 91:

By the mid-60s, the importance of pipelining in increasing machine performance was clear to IBM. An article in the January 1967 IBM Journal [AND67, p 8] presents the organizational philosophy utilized in IBM's highest performance computer, the System/360 Model 91. The first section of the paper deals with the development of the assembly-line processing approach adopted for the Model 91.

The assembly-line approach (pipelining) had become the first topic when explaining why IBM's fastest machine was fast. The goal for the 360/91 was a performance increase of 10 to 100 times that of the 7090. The circuit and hardware advance could only provide a fourfold increase, so organizational techniques would have to provide the remaining improvement. The authors state that the primary organizational objective for a high performance CPU is concurrency -- the parallel execution of different instructions. [...] it is desired to "overlay" the separate instruction functions to the greatest degree possible.

The 360/91 CPU loads the instruction fetch buffers with eight double words. This pre-fetch hides the instruction access time for straight-line (no-branching) programs. In order to minimize the disruption of the flow of instructions when a branch is encountered, the CPU fetches two double words down the target path of the branch as a hedge against taking the branch. Maximum performance is achieved when the branch is not taken.

With smooth-flowing instruction streams where the pipeline can be kept full, programs can execute at 100 times the speed of the 7090, about 25 times the performance increase due to faster hardware. For applications that do not flow smoothly, performance diminishes to about 10 times the speed of the 7090 ([AND67], p12). This is still 2.5 times better than the performance increase due to faster

hardware.

Performance is improved in the IBM 360/91 by a factor of 2.5 to 25 due to pipelining. Pipelining is facilitated by the division of float-point operations and fixed-point operations into separate units. As long as no dependencies exist, the execution is not necessarily in the order in which the instructions were programmed.

The fixed-point unit executes instructions serially [HWA84]. The floating-point unit has additional concurrency. Here again, independent instructions may be executed out of order. Interrupts are a major bottleneck to performance in the pipelined architecture. Strict adherence to a sequential flow of instruction would reduce performance. Another approach would be to set aside sufficient information to properly recover from any interrupt which would occur. This approach was too complex and costly for the designers. The solution was to compromise the architecture by allowing the CPU to continue to execute instructions in the pipeline after the interrupt, even if one of the instruction being executed caused the interrupt. This is termed the "imprecise interrupt".

Interrupts associated with an instruction which can be uncovered during instruction decoding are precise. Only those interrupts which result from address, storage, or execution are imprecise. Considered as a whole, the architecture of the 360/91 was a major step forward. The concepts outlined in [AND67] are frequently cited in subsequent works on pipelining. For instance, Hennessy and Patterson [HEN94] state these concepts. Many of the ideas in the 360/91 faded from use for nearly 25 years before being broadly employed in the 1990s.

The IBM 7030 Stretch Computer

Introduction to Project Stretch

In late 1954, a project was started at IBM to design a new computer. The entire field was still new, there had been little experience on which to base the design of such computers as the 704 (a large-scale scientific computer) and the 705 (a business data processing computer). With the experience gained in making those machines, improvements in design became apparent. Furthermore, it appeared possible to soon produce computers with transistors and diodes replacing vacuum tubes.

The project was set forth to design and build a computer with a goal of reaching an over-all performance of 100 times that of the 704, then the faster computer available. This would make available the best technology that could be achieved by straining the technical resources. Hence the name Project Stretch. The project introduced a number of new concepts into computer architecture and brought forth, new terms into the language of computers. One of these terms was computer architecture [CAM62, p 5].

Concurrency within the CPU

During the design of the Stretch, it appeared that a factor-of-6 improvement in memory speed and a factor-of-10 improvement in basic circuit speed were the best to be achieved. Since the goal was a factor-of-100 overall improvement, the design had to provide for concurrent operation wherever possible [CAM62, p 11,202-6]. Up to eleven successive instructions may be in the registers of the central processing unit at various stages of execution.

The CPU processes and executes instructions with a high degree of overlap of internal functions. Two instruction words, which often are four half-word instructions, and the operands for four more instructions can be simultaneously fetched. At the same time, two more instructions can be decoded and another executed. The CPU is micro-coded, with the instruction unit having its own instructions to execute, its own small memory and its own arithmetic unit. This micro-coded instruction unit is pipelined as well, with as many of six instructions being in various stages of progress. By pre-fetching and examining instructions before execution, it is possible to determine the address of operands required. The look-ahead unit would do this.

Performance Gains in the Stretch Computer

As stated previously, the goal of the designers of the 7030 Stretch was to make the computer 100 times faster than the IBM 704. The 7030 used memory that was 6 times as fast as that in the 704 (2 usec vs 12 usec) and circuits that were about 10 times as fast as the 704. [CAM62, p 202]

They did not meet their design objective completely. With small programs that fit in the 704's word size, the performance increase on the 7030 was a factor of 70. On larger programs that strained the 704, the performance ratio was greater than 100.

Therefore, with about a factor-of-10 increase in CPU and memory speed, the 7030 achieved about a factor-of-100 increase in overall performance. While pipelining was not the only architectural advance, it was the most significant one. In conclude pipelining yielded a factor-of-10 performance improvement in the Stretch computer.

The original Stretch computer was delivered in 1961. Fewer than ten more were built. But the ideas were not lost. The Stretch was to be highly influential in the design of IBM's next family of computers, the System 360. But for the stretch it was slower than expected. [WebSite1], [WebSite2].

Motorola 68060

The Motorola 68060 is a fully pipelined super-scalar processor. The 68060 allows simultaneous execution of two integer instructions (or 1 integer and 1 float instruction) and one branch during each clock cycle. A branch cache allows most branches to execute in zero cycles. It contains a 4-stage instruction fetch pipeline, and two 6-stage pipelines for the primary operand execution and the secondary operand execution. [TAB95, Ch12]

The 68060 allows simultaneous execution of two integer instructions during one clock cycle.

IFP (Instruction Fetch Pipeline) stages

1. IAG - Instruction Address Generation
2. IC - Instruction cache access
3. IED - Instruction Early Decode
4. IB - Instruction Buffer

OEP (Operand Execution Pipeline) stages

1. DS - Decode and Select instructions
2. AG - Address Generation of the operand

3. OC - Operand Cache access
4. EX - Execute
5. DA - Data Available
6. ST – Store

Intel Pentium

The Pentium is a super-scalar fully pipelined microprocessor. A super-scalar processor has the ability to process more than one instruction per clock cycle. The Pentium has two execution pipes (U and V) so it is a super-scalar level 2 processor.

The Pentium has dual internal caches, for both code and data, and dual TLBs. The TLB is the Translation Lookaside Buffer which caches the virtual page number to the physical page number. This facilitates efficient handling of the pipeline.

The Pentium pre-fetches as much as 32 bytes of instruction. It employs branch-prediction, a technique that attempts to infer the proper next instruction address, in order to keep the pipeline from stalling. In the Pentium, a super-scalar level 2 processor, two instructions are fetched and decoded simultaneously.

Instruction pipeline

1. PF - Fetch and Align instruction
2. D1 - Decode Instruction, Generate Control Word
3. D2 - Decode Control Word, Generate memory address
4. E - Access data cache or Calculate ALU Result
5. WB - Write Result

Floating-Point Pipeline

1. PF – Pre-fetch
2. D1 - First Decode

3. D2 - Second Decode
4. E - Operand Fetch
5. X1- First execute
6. X2 - Second Execute
7. WF - Write Float
8. ER - Error Reporting

Conclusion

Part of the excitement in working in the computer industry is the rate of progress. In November 1995, Intel released the P6, the Pentium Pro microprocessor. None of the library books consulted had any reference to this advanced CPU. Information was gathered from the Intel World Wide Web site and [HEN94]

The P6 (Pentium Pro) is a superscalar level 3 processor. It has three pipelines and is capable of executing three integer instructions per clock cycle. Further more, it employs speculative execution to predict program flow and to execute ahead o



Chapter 2

OVERVIEW OF PROCESSOR ARCHITECTURE AND PIPELINES

This section provides an overview of the pipelines and architectural features of the Pentium® and dynamic execution (P6-family) processors with MMX™ technology. By understanding how the code flows through the pipeline of the processor, you can better understand why a specific optimization will improve the speed of your code. Additionally, it will help you to schedule and optimize your application for high performance.

2.1 Pipelines of Superscalar (Pentium® Family) and Dynamic Execution (P6-Family) Architectures

2.1.1 SUPERSCALAR (PENTIUM® FAMILY) PIPELINE

The Pentium processor is an advanced superscalar processor. It is built around two general-purpose integer pipelines and a pipelined floating-point unit, allowing the processor to execute two integer instructions simultaneously. A software-transparent dynamic branch-prediction mechanism minimizes pipeline stalls due to branches. Pentium processors with MMX technology add additional stages to the pipeline. The integration of the MMX Technology pipeline with the integer pipeline is very similar to that of the floating-point pipe.

Pentium processors can issue two instructions every clock cycle, one in each pipe. The first logical pipe is referred to as the "U" pipe, and the second as the "V" pipe. During decoding of any given instruction, the next two instructions are checked, and, if possible, they are issued such that the first one executes in the U-pipe and the second in the V-pipe. If it is not possible to issue two instructions, then the next instruction is issued to the U-pipe and no instruction is issued to the V-pipe.

When instructions execute in the two pipes, their behavior is exactly the same as if they were executed sequentially. When a stall occurs, successive instructions are not allowed to pass the stalled instruction in either pipe. Figure 2-1 shows the pipelining structure for this scheme:



Figure 2-1. MMX™ Technology Pipeline Structure

Pentium processors with MMX technology add an additional stage to the integer pipeline. The instruction bytes are prefetched from the code cache in the prefetch (PF) stage, and they are parsed into instructions in the fetch (F) stage. Additionally, any prefixes are decoded in the F stage.

Instruction parsing is decoupled from the instruction decoding by means of an instruction First In, First Out (FIFO) buffer, which is situated between the F and Decode 1 (D1) stages. The FIFO has slots for up to four instructions. This FIFO is transparent; it does not add additional latency when it is empty.

During every clock cycle, two instructions can be pushed into the instruction FIFO (depending on availability of the code bytes, and on other factors such as prefixes). Instruction pairs are pulled out of the FIFO into the D1 stage. Since the average rate of instruction execution is less than two per clock, the FIFO is normally full. As long as the FIFO is full, it can buffer any stalls that may occur during instruction fetch and parsing. If such a stall occurs, the FIFO prevents the stall from causing a stall in the execution stage of the pipe. If the FIFO is empty, then an execution stall may result from the pipeline being "starved" for instructions to execute. Stalls at the FIFO entrance may result from long instructions or prefixes (see Sections 3.2.3 and 3.4.2).

The following chart details the MMX technology pipeline on superscalar processors and the conditions in which a stall may occur in the pipeline.



Figure 2-2. MMX™ Technology Instruction Flow in a Pentium® Family Processor with MMX Technology

Table 2-1 details the functional units, latency, throughput, and execution pipes for each type of MMX technology instruction.

Table 2-1. MMX™ Technology Instructions and Execution Units

Operation	Number of Functional Units	Latency	Throughput	Execution Pipes
ALU	2	1	1	U and V
Multiplexer	1	3	1	U or V
Shift/pack/unpack	1	1	1	U or V
Memory access	1	1	1	U only
Integer register access	1	1	1	U only

- The Arithmetic Logic Unit (ALU) executes arithmetic and logic operations (that is, add, subtract, xor, and).
- The Multiplier unit performs all multiplication operations. Multiplication requires three cycles but can be pipelined, resulting in one multiplication operation every clock cycle. The processor has only one multiplier unit which means that multiplication instructions cannot pair with other multiplication instructions. However, the multiplication instructions can pair with other types of instructions. They can execute in either the U- or V-pipes.
- The Shift unit performs all shift, pack and unpack operations. Only one shifter is available so shift, pack and unpack instructions cannot pair with other shift unit instructions. However, the shift unit instructions can pair with other types of instructions. They can execute in either the U- or V-pipes.
- MMX Technology Instructions that access memory or integer registers can only execute in the U-pipe and cannot be paired with any instructions that are not MMX Technology instructions.
- After updating an MMX technology register, two clock cycles must pass before that MMX technology register can be moved to either memory or to an integer register.

Information on pairing requirements can be found in Section 3.3.

Additional information on instruction format can be found in the *Intel Architecture MMX™ Technology Programmer's Reference Manual*, (Order Number 243007).

2.1.2. DYNAMIC EXECUTION (P6-FAMILY) PIPELINE

P6-family processors use a Dynamic Execution architecture that blend out-of-order and speculative execution with hardware register renaming and branch prediction. These processors feature an in-order issue pipeline, which breaks Intel386™ processor macroinstructions up into simple, micro-operations called micro-ops (or uops), and an out-of-order, superscalar processor core, which executes the micro-ops. The out-of-order core of the processor contains several pipelines to which integer, jump, floating-point, and memory execution units are attached. Several different execution units may be clustered on the same pipeline: for example, an integer address logic unit and the floating-point execution units (adder, multiplier, and divider) share a pipeline. The data cache is pseudo-dual ported via interleaving, with one port dedicated to loads and the other to stores. Most simple operations (integer ALU, floating-point add, even floating-point multiply) can be pipelined with a throughput of one or two operations per clock cycle. Floating-point divide is not pipelined. Long latency operations can proceed in parallel with short latency operations.

The P6-family pipeline is comprised of three parts: the In-Order Issue Front-end, the Out-of-Order Core and the In-Order Retirement unit. Details about the In-Order Issue Front-end follow below.

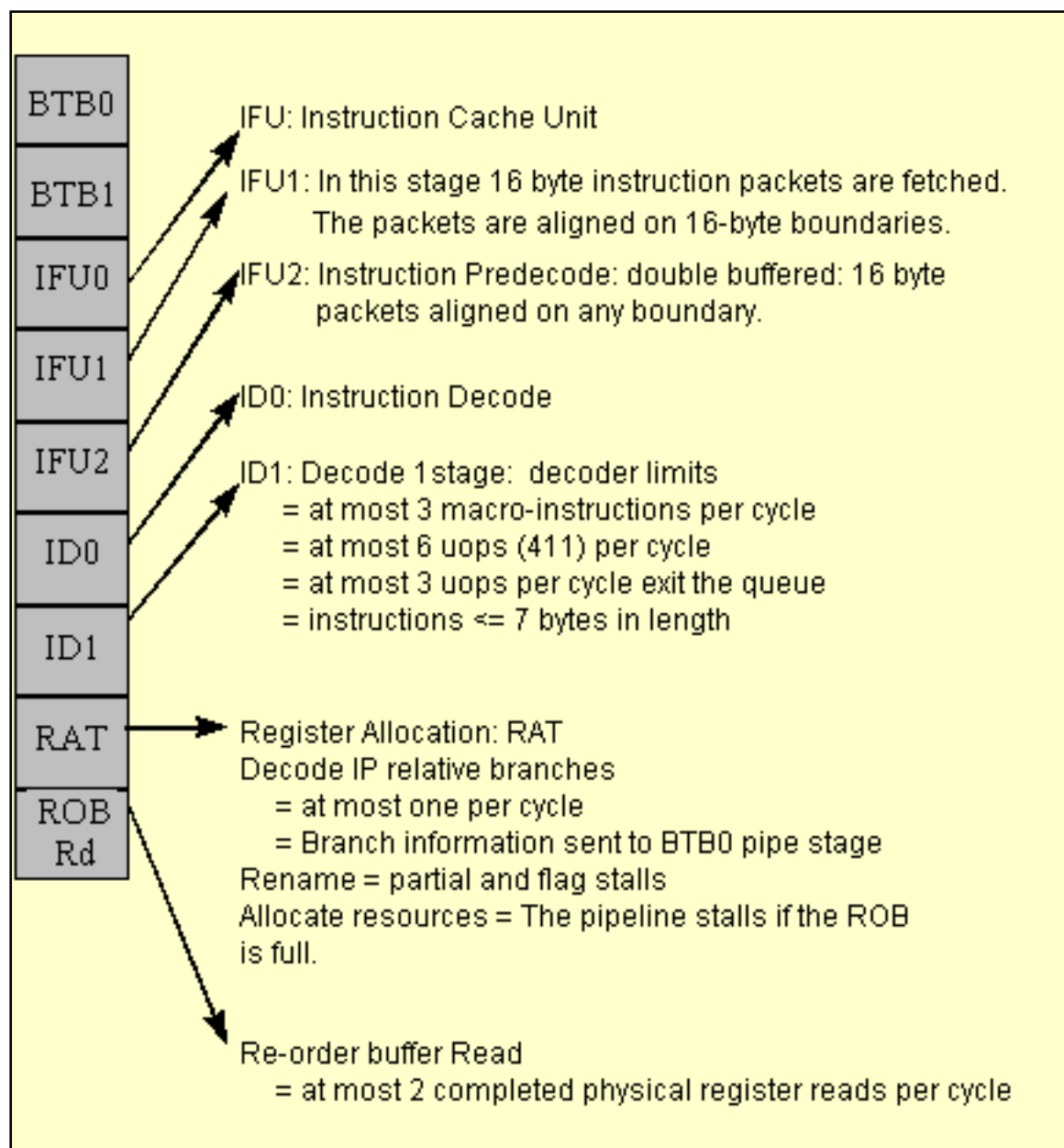


Figure 2-3. Out-Of-Order Core and Retirement Pipeline

Since the dynamic execution processors execute instructions out of order, the most important consideration in performance tuning is making sure enough micro-ops are ready for execution. Correct branch prediction and fast decoding are essential to getting the most performance out of the In-Order Front-End. Branch prediction and the branch target buffer are detailed in Section 2.3. Decoding is discussed below.

During every clock cycle, up to three Intel Architecture macro instructions can be decoded in the ID1 pipestage. However, if the instructions are complex or are over seven bytes then the decoder is limited to decoding fewer instructions.

The decoders can decode:

1. Up to three macro-instructions per clock cycle.
2. Up to six micro-ops per clock cycle.
3. Macro-instructions up to seven bytes in length.

P6-family processors have three decoders in the D1 pipestage. The first decoder is capable of decoding one IA macro-instruction of four or fewer micro-ops in each clock cycle. The other two decoders can each decode an IA instruction of one micro-op in each clock cycle. Instructions composed of more than four micro-ops will take multiple cycles to decode. When programming in assembly language, scheduling the instructions in a 4-1-1 micro-op sequence increases the number of instructions that can be decoded each clock cycle. In general:

- Simple instructions of the register-register form are only one micro-op.
- Load instructions are only one micro-op.
- Store instructions have two micro-ops.
- Simple read-modify instructions are two micro-ops.
- Simple instructions of the register-memory form have two to three micro-ops.
- Simple read-modify write instructions are four micro-ops.
- Complex instructions generally have more than four micro-ops, therefore they will take multiple cycles to decode.

For the purpose of counting micro-ops, MMX technology instructions are simple instructions. See *Optimizations for Intel's 32-bit Processors*, Application Note AP-526 (Order Number 242816), Appendix D for a table that specifies the number of micro-ops for each instruction in the Intel Architecture instruction set.

Once the micro-ops are decoded, they will be issued from the In-Order Front-End into the Reservation Station (RS), which is the beginning pipestage of the Out-of-Order core. In the RS, the micro-ops wait until their data operands are available. Once a micro-op has all data sources available, it will be dispatched from the RS to an execution unit. If a micro-op enters the RS in a data-ready state (that is, all data is available), then the micro-op will be immediately dispatched to an appropriate execution unit, if one is available. In this case, the micro-op will spend very few clock cycles in the RS. All of the execution units are clustered on ports coming out of the RS. Once the micro-op has been executed it returns to the ROB, and waits for retirement. In this pipestage, all data values are written back to memory and all micro-ops are retired in-order, three at a time. The figure below provides details about the Out-of-Order core and the In-Order retirement pipestages.

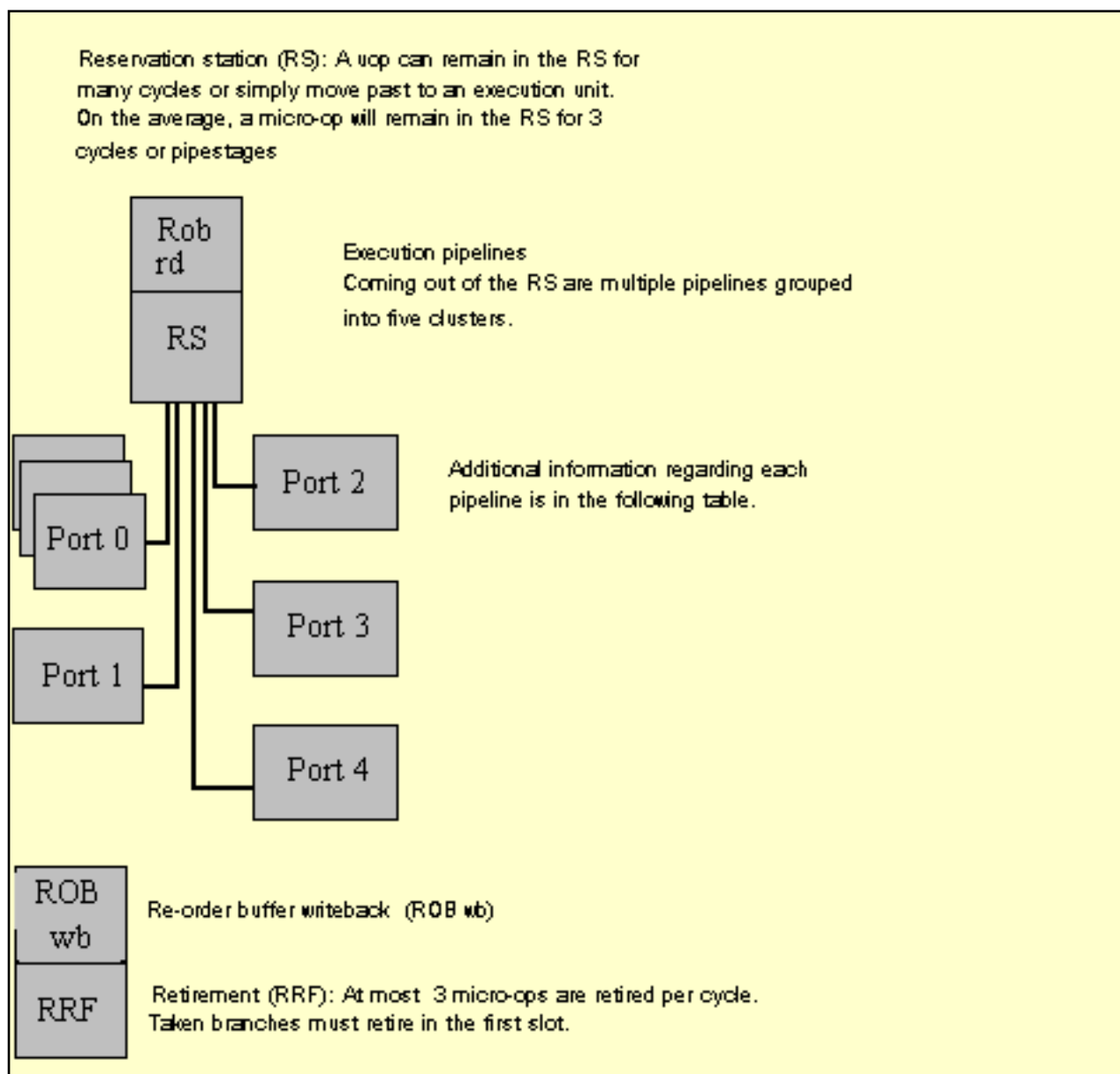


Figure 2-4. Out-Of-Order Core and Retirement Pipeline

Table 2-2. Dynamic Execution (P6-Family) Processor Execution Unit Pipelines

Port	Execution Units	Latency/Thruput
0	Integer ALU Unit LEA instructions Shift instructions	Latency 1, Thruput 1/cycle Latency 1, Thruput 1/cycle Latency 1, Thruput 1/cycle
	Integer Multiplication instruction	Latency 4, Thruput 1/cycle
	Floating-Point Unit FADD instruction FMUL FDIV Unit	Latency 3, Thruput 1/cycle Latency 5, Thruput 1/2cycle1,2 Latency long and data dep., Thruput non-pipelined
	MMX™ Technology ALU Unit	Latency 1, Thruput1/cycle

	MMX Technology Multiplier UnitFDIV	Latency 3, Thruput 1/cycle
1	Integer ALU Unit	Latency 1, Thruput 1/cycle
	MMX Technology ALU Unit	Latency 1, Thruput 1/cycle
	MMX Technology Shifter Unit	Latency 1, Thruput 1/cycle
2	Load Unit	Latency 3 on a cache hit, Thruput 1/cycle4
3	Store Address Unit	Latency 3 (not applicable) Thruput 1/cycle3
4	Store Data Unit	Latency 1 (not applicable) Thruput 1/cycle

Notes:

1. The FMUL unit cannot accept a second FMUL within the cycle after it has accepted the first. This is NOT the same as only being able to do FMULs on even clock cycles.
2. FMUL is pipelined one every two clock cycles. One way of thinking about this is to imagine that a P6-family processor has only a 32x32->32 multiply pipelined.
3. Store latency is not all that important from a dataflow perspective. The latency that matters is with respect to determining when they can retire and be completed. They also have a different latency with respect to load forwarding. For example, if the store address and store data of a particular address, for example 100, dispatch in clock cycle 10, a load (of the same size and shape) to the same address 100 can dispatch in the same clock cycle 10 and not be stalled.
4. A load and store to the same address can dispatch in the same clock cycle.

2.2 Caches

The on-chip cache subsystem of processors with MMX technology consists of two 16 K four-way set associative caches with a cache line length of 32 bytes. The caches employ a write-back mechanism and a pseudo-LRU replacement algorithm. The data cache consists of eight banks interleaved on four-byte boundaries.

On Pentium processors with MMX technology, the data cache can be accessed simultaneously from both pipes, as long as the references are to different cache banks. On the dynamic execution (P6-family) processors, the data cache can be accessed simultaneously by a load instruction and a store instruction, as long as the references are to different cache banks. The delay for a cache miss on the Pentium processor with MMX technology is eight internal clock cycles. On dynamic execution processors with MMX technology the minimum delay is ten internal clock cycles.

2.3 Branch Target Buffer

Branch prediction for Pentium and dynamic execution processors with MMX technology is functionally identical except for one minor exception which will be discussed in Section 2.3.1.

The Branch Target Buffer (BTB) stores the history of the previously seen branches and their targets. When a branch is prefetched, the BTB feeds the target address directly into the Instruction Fetch Unit (IFU). Once the branch is executed, the BTB is updated with the target address. Using the branch target buffer, branches that have been seen previously are dynamically predicted. The branch target buffer prediction algorithm includes pattern matching and up to four prediction history bits per target address. For example, a loop which is four iterations long should have close to 100% correct prediction. Adhering to the following guideline will improve branch prediction performance:

Program conditional branches (except for loops) so that the most executed branch immediately follows the branch instruction (that is, fall through).

Additionally, processors with MMX technology have a Return Stack Buffer (RSB), which can correctly predict return addresses for procedures that are called from different locations in succession. This increases further the benefit of unrolling loops which contain function calls, and removes the need to in-line certain procedures.

2.3.1 CONSECUTIVE BRANCHES

On the Pentium processor with MMX technology, branches may be mispredicted when the last byte of two branch instructions occur in the same aligned four byte section of memory, as shown in the figure below.

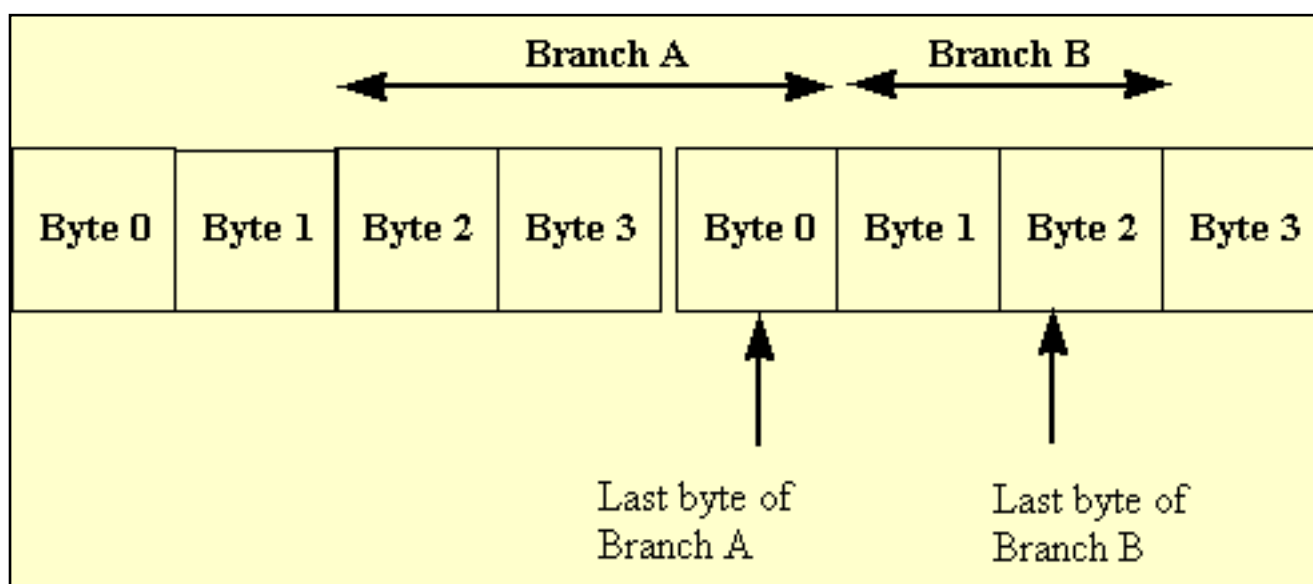


Figure 2-5. Consecutive Branch Example

This may occur when there are two consecutive branches with no intervening instructions and the second instruction is only two bytes long (such as a jump relative +/- 128).

To avoid a misprediction in these cases, make the second branch longer by using a 16-bit relative displacement on the branch instruction instead of an 8-bit relative displacement.

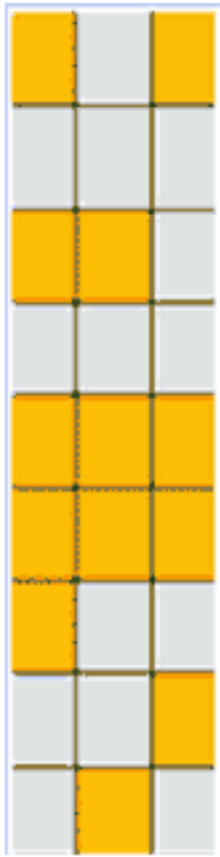
2.4 Write Buffers

Processors with MMX technology have four write buffers (versus two in Pentium processors without MMX

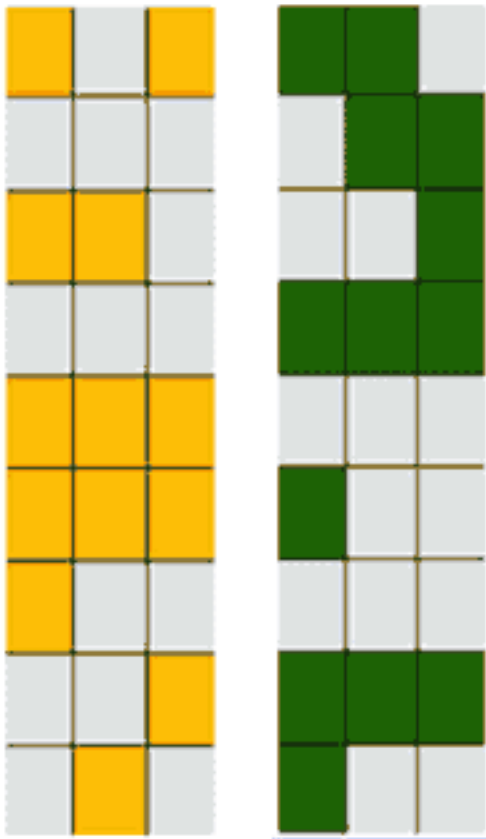
technology). Additionally, the write buffers can be used by either the U-pipe or the V-pipe (versus one corresponding to each pipe in Pentium processors without MMX technology). Performance of critical loops can be improved by scheduling the writes to memory; when you expect to see write misses, you should schedule the write instructions in groups no larger than four, then schedule other instructions before scheduling further write instructions.

[Legal Information](#) © 1997 Intel Corporation

Superskalare Ausführung



Multiprozessor-basierte Ausführung



Multithreading-CPU-basierte Ausführung



1 CPU

Hyper-threading

From Wikipedia, the free encyclopedia.

Hyper-threading is Intel's trademark for their implementation of the simultaneous multithreading technology on the Pentium 4 microarchitecture. It is basically a more advanced form of Super-threading that first debuted on the Intel Xeon processors and later added to Pentium 4 processors. The technology improves processor performance under certain workloads by providing useful work for execution units that would otherwise be idle, for example during a cache miss.

The advantages of Hyper-Threading are listed as improved support for multi-threaded code, allowing multiple threads to run simultaneously, improved reaction and response time, and increased number of users a server can support.

Hyper-Threading works by duplicating certain sections of the processor—those that store the architectural state—but not duplicating the main execution resources. This allows a Hyper-Threading equipped processor to pretend to be two "logical" processors to the host operating system, allowing the operating system to schedule two threads or processes simultaneously. Where execution resources in a non-Hyper-Threading capable processor are not used by the current task, and especially when the processor is stalled, a Hyper-Threading equipped processor may use those execution resources to execute the other scheduled task. (Reasons for the processor to stall include a cache miss, a branch misprediction and waiting for results of previous instructions before the current one can be executed.)

Except for its performance implications, this innovation is transparent to operating systems and programs. All that is required to take advantage of Hyper-Threading is SMP support in the Operating System, as the logical processors appear as standard separate processors.

However, it is possible to optimise operating system behaviour on Hyper-Threading capable systems, such as the Linux techniques discussed in Kernel Traffic (<http://www.kerneltraffic.org/>)

kernel-traffic/kt20020902_182.html). (One such optimisation concerns a dual-processor system where both processors are capable of Hyper-Threading. The cost of moving a process from one logical processor to another on the same physical processor is almost nothing, whereas processor affinity provides significant reasons to keep processes on the same physical processor.)

According to Intel, the first implementation only used an additional 5% of the die area over the "normal" processor, yet yielded performance improvements of 15-30%.

External Links

- Intel's High level overview (*<http://www.intel.com/technology/hyperthread/>*)
- Hyper-Threading Technology Architecture and Microarchitecture (*http://www.intel.com/technology/itj/2002/volume06issue01/art01_hyper/p01_abstract.htm*) - a more technical description of Hyper-Threading

-
- This page was last modified 14:21, 25 May 2004.
 - All text is available under the terms of the GNU Free Documentation License (see **Copyrights** for details).

GIRLSGOTTECH.ORG



☒ Articles ☐ Forums

Contents

- Home
- News
- Reviews
- Editorials
- Daily Column
- Home Theater
- Forums
- Community

- About Us
- Policies

Latest News

- InFocus Launches ScreenPlay 4805 Wide Screen Projector
- ATi Launches X800 Series of Graphics Accelerators
- Nvidia Launches GeForce 6 Series of Graphics Accelerators
- Crucial Announces DDR2 Memory Availability
- Microsoft and Sun Microsystems Settle Outstanding Litigation
- Olympus E-1 and Olympus E-System Receive DIWA Awards
- ASUS Introduces DVD Rewriter with 8x DVD+-R, 4x DVD+-RW

Latest Topics

- Old Mouse doesn't work - anybody recommend a new one????
- Some questions on the AMD64
- Western Digital WD1600BB Making Strange Noises
- On A Lighter Note!
- Problems with my Ati Radeon 9600xt
- HELP!!! ASUS A7N266-VM
- Sony Vaio Notebook is dead NO Power

[More >>](#)

Advertisement

Article Spotlights



Popular Articles

You Are Here:

[/ Home / Reviews /](#)

Hyper-Threading On The Desktop

Nov 14, 2002, 02:00 AM

SMT and SMP

By: [Sander Sassen](#)

Stating that previous, conventional CPUs do not offer true-multitasking abilities might sound like a bold statement, especially since Windows is all about multitasking. In essence conventional CPUs use various techniques to mimic multitasking behavior or shuffle instructions around and so does the Operating System. Modern CPUs for example, use out-of-order execution, OOE, which enables the CPU to make maximum use of processor resources.

An OOE processor combines multiple instruction sequences, meant to be executed in a specific order and reschedules them so that they can be executed with the highest efficiency. After execution they're arranged back into their original order and the results are then written to main memory or are available for the next instruction sequence. To the user this process is transparent, only the CPU knows in what order the instructions were actually executed.

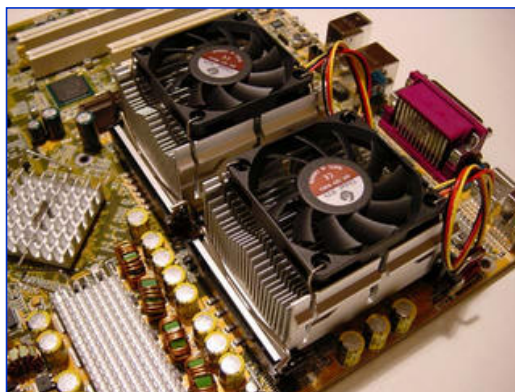


Fig 2. A typical SMP configuration featuring two Intel Xeon processors that feature Hyper-Threading.

Something similar actually happens when running Windows. To the user it appears that multiple programs are running at the same time, but in reality there's only one program active at any given instant. A conventional single-threaded CPU can only execute one of these programs at a time and thus the Operating System mimics multitasking behavior by giving each program a piece of the available processor time and then rapidly alternating between the pieces. What you change when you give a certain program 'Above Normal' priority in the taskmanager is simply giving it more execution time.

With SMT, Simultaneous Multi Threading, and SMP, Symmetric Multi Processing, we have two ways of going beyond the limitations of a conventional single-threaded processor. SMP is simply the addition of another single-threaded processor and results in a configuration being able to execute two threads simultaneously; SMT is the presence of another logical processor within the same physical processor and again the ability to execute two threads simultaneously.

From that perspective it would mean that SMP and SMT are somewhat comparable and that Hyper-Threading is an alternative to SMP. Unfortunately this is not entirely accurate, although Hyper-Threading tackles a lot of the multitasking hurdles and does offer true-multitasking support there's more going on in the processor' core that needs attention.

Advertisement



Related Articles

- [Computex 2004 Taiwan, wrapup and more](#)
- [Computex 2004, June 2nd, Abit, SiS and ALi/ULI](#)
- [Computex 2004, June 1st, Nvidia, S3 and iwill](#)
- [AMD roadmap, socket to me baby?](#)
- [A look at 802.11g, Compex and X-Micro](#)
- [Server downtime, SCSI controller woes](#)
- [InFocus Launches ScreenPlay 4805 Wide Screen Projector](#)

Tools

- [Print This Article](#)

Newsletter

A weekly newsletter featuring an editorial and a roundup of the latest articles, news and other interesting topics.

Please enter your email address below and click [Subscribe](#).

About Us



America the beautiful.

So why speed through it?

Driving calmly saves gas and reduces our dependence on oil.

Click for more ways to get green.

e
ENVIRONMENTAL DEFENSE

get green



[Next >>](#)

1. [Introduction](#)
2. [SMT and SMP](#)
3. [Scheduling and Caching](#)
4. [Proformance or Performance](#)
5. [Hardware and Software](#)
6. [Benchmarks](#)
7. [Benchmarks: Multitasking](#)
8. [Benchmarks: Multitasking Cont.](#)
9. [Summary](#)
10. [Appendix](#)

[Discuss This Article](#) - If you have any questions, comments or suggestions about the article and/or its contents please leave your comments here and we'll do our best to address any concerns.

[Rate This Product](#) - If you have first hand experience with this product and would like to share your experience with others please leave your comments here.

- [Advertising](#)
- [Disclaimer](#)
- [Fair Use Policy](#)
- [Privacy Policy](#)
- [Who We Are](#)
- [Link To Us](#)



We want YOU to take this survey!

		Data	
		Single	Multiple
Instruction	Multiple	MISD	MIMD
	Single	SISD	SIMD

Message Passing Interface

From Wikipedia, the free encyclopedia.

The **Message Passing Interface** (MPI) is a computer communications protocol. It is a *de facto* standard for communication among the nodes running a parallel program on a distributed memory system. MPI is a library of routines that can be called from Fortran and C programs. MPI's advantage over older message passing libraries is that it is both portable (because MPI has been implemented for almost every distributed memory architecture) and fast (because each implementation is optimized for the hardware it runs on).

The most common implementation in use is MPICH.

See also

- Occam
- Linda
- PVM
- Calculus of Communicating Systems
- Calculus of Broadcasting Systems

External links

- MPICH Home Page (*<http://www-unix.mcs.anl.gov/mpi/mpich/>*)
- MPI Forum (*<http://www.mpi-forum.org/>*)
- Citations from CiteSeer (*<http://citeseer.org/cs?q=Message+and+Passing+and+Interface>*)

This article (or an earlier version of it) contains material from FOLDOC, used with permission.

-
- This page was last modified 02:23, 2 May 2004.
 - All text is available under the terms of the GNU Free Documentation License (see **Copyrights** for details).

Amdahl's law

From Wikipedia, the free encyclopedia.

Amdahl's law, named after computer architect Gene Amdahl, is concerned with the speedup achievable from an improvement to a computation that affects a proportion P of that computation where the improvement has a speedup of S . For example, if an improvement can speedup 30% of the computation, P will be 0.3; if the improvement makes the portion affected twice as fast, S will be 2. Amdahl's law states that the overall speedup of applying the improvement will be

$$\frac{1}{(1 - P) + \frac{P}{S}}.$$

To see how this formula was derived, assume that the running time of the old computation was 1, for some unit of time. The running time of the new computation will be the length of time the unimproved fraction takes (which is $1-P$) plus the length of time of the improved fraction takes. The length of time for the improved part of the computation is the length of the old running time divided by the speedup, making the length of time of the improved part P/S . The final speedup is computed by dividing the old running time by the new running time, which is what the above formula does.

Amdahl's law is a demonstration of the law of diminishing returns: while one could speed up part of a computer a hundred-fold or more, if it only affects 12% of the overall task the best the

speedup could possibly be is $\frac{1}{1 - 0.12} = 1.136$ times faster.

In the special case of parallelization, Amdahl's law states that if F is the fraction of a calculation that is sequential (i.e. cannot benefit from parallelisation, and $(1-F)$ is the fraction that can be parallelised, then the maximum speedup that can be achieved by using N processors is

$$\frac{1}{F + \frac{1-F}{N}}.$$

In the limit, as *N* tends to infinity, the maximum speedup tends to $1/F$. In practice, price/performance ratio falls rapidly as *N* is increased once $(1-F)/N$ is small compared to *F*.

As an example, if *F* is only 10%, the problem can be sped up by only a maximum of a factor of 10, no matter how large the value of *N* used. For this reason, parallel computing is only useful for either small numbers of processors, or problems with very low values of *F*: so-called embarrassingly parallel problems. A great part of the craft of parallel programming consists of attempting to reduce *F* to the smallest possible value.

References:

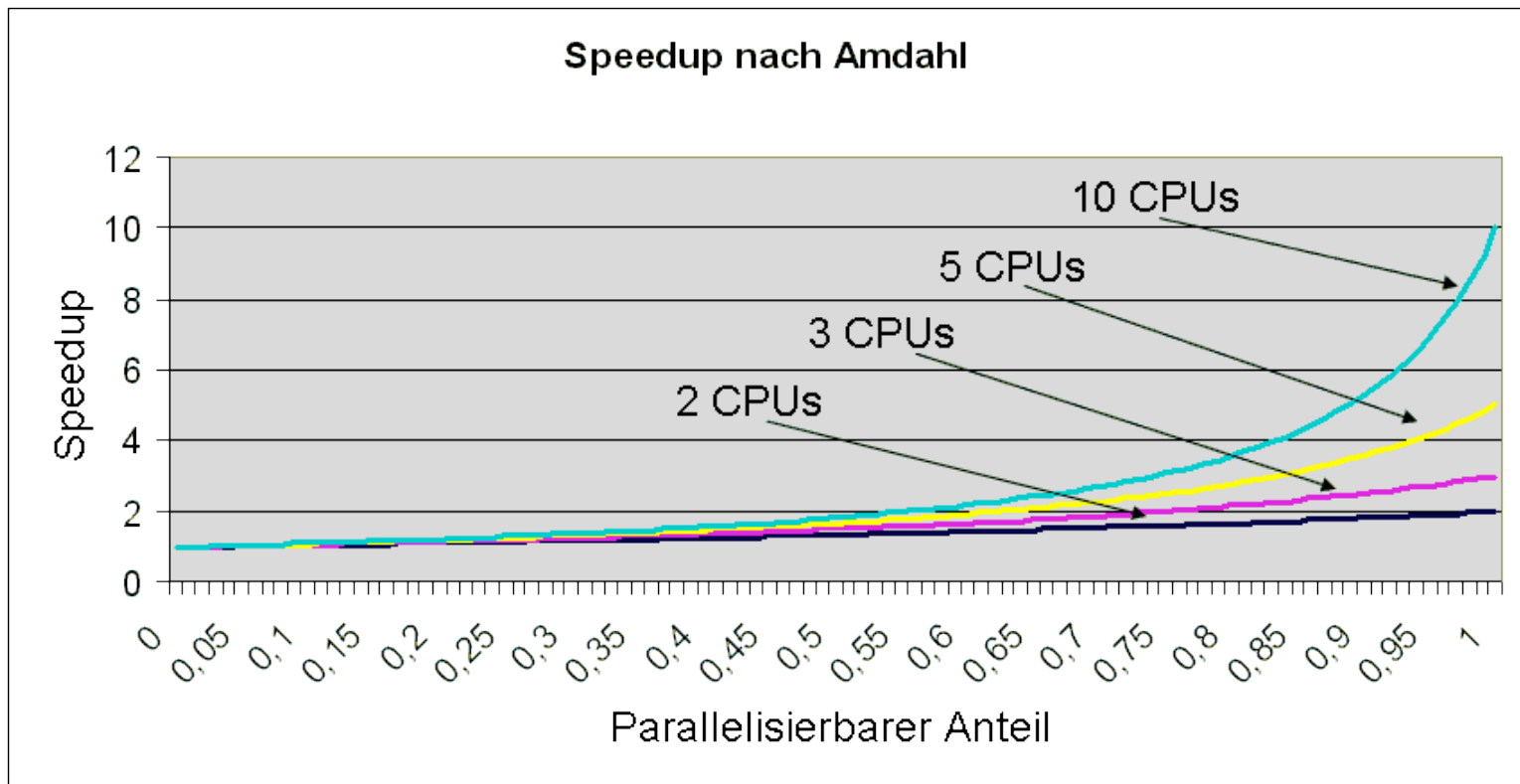
- Gene Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings, (30), pp. 483-485, 1967.

External links

- Reevaluating Amdahl's Law (*<http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html>*)

-
- This page was last modified 17:53, 5 May 2004.
 - All text is available under the terms of the GNU Free Documentation License (see **Copyrights** for details).

$$S := \frac{1}{p_{\text{seriell}} + \frac{p_{\text{parallel}}}{n_{\text{CPU}}}}$$



▲ Vorlesung Java

▼ 1 Einführung in Java

▼ 1.1 Was ist Java?

▼ 1.2 Entstehungsgeschichte

▼ 1.3 Die Java-Plattform

▼ 1.4 Vom Quellcode zum lauffähigen Programm

▼ 2 Syntax und Semantik der Programmiersprache Java

▼ 2.1 C, C++, C# und Java

▼ 2.2 Grundstrukturen

▼ 2.2.1 Programmaufbau

▼ 2.2.2 Einfache Datentypen

▼ 2.2.3 Operatoren

▼ 2.3 Kontrollstrukturen

▼ 2.3.1 Selektion und Mehrfachselektion -- das if und case-Statement

▼ 2.3.2 Iteration -- for-, do-while-Schleifen

▼ 2.3.3 Ausnahmen und ihre Behandlung -- Exception Handling

▼ 2.3.4 Zwangsbedingungen

▼ 2.4 Von komplexen zu objektorientierten Datenstrukturen

▼ 2.4.1 Arrays

▼ 2.4.2 Klassen

▼ 2.4.3 Attribute

▼ 2.4.4 Operationen und Methoden

▼ 2.4.5 Aufzählungstypen

▼ 2.4.6 Wrapper-Typen

▼ 2.4.7 Boxing / Unboxing

▼ 2.4.8 Vererbung

▼ 2.4.9 Schnittstellen

▼ 2.4.10 Pakete

▼ 3 Die Java-Plattform

▼ 3.1 Die Laufzeitumgebung

▼ 3.1.1 Garbage Collection

▼ 3.1.2 Virtuelle Maschine

▼ 3.2 Die Java API und weiterführende Themen

▼ 3.2.1 Ein-/Ausgabe -- Streams

▼ 3.2.2 Threads und Nebenläufigkeit

▼ 3.2.3 Applets

▼ 3.2.4 Collection API

▼ 3.2.5 Generics

▼ 3.2.6 Reflection API

▼ 3.2.7 Abstract Windowing Toolkit (AWT)

▼ 3.2.8 Swing

▲ 1 Einführung in Java

▲ 1.1 Was ist Java?

- **Laut Brockhaus:**
 - von staatseigenen indones[ischen] Plantagen auf Java stammender Blatt-Tabak, der je nach Qualität als Deckblatt, Umblatt oder Einlage bei der Zigarrenherstellung verwendet wird.
Aus: Brockhaus --- Die Enzyklopädie, Bd. 11, 20. Aufl., 1997, p. 149.
 - **Java, Jawa** früher **Djava** [...], kleinste, aber bedeutendste der Großen Sundainseln, Indonesien, 1 060 km lang, 55-195 km breit, 118 000 km², (1990) 107,58 Mio. Ew.; verwaltungsmäßig (einschließlich der Insel Madura 132 186 km²) in drei Prov., die Sonderregion Yogyakarta und den hauptstadtsonderbezirk Jakarta gegliedert. Die Insel ist überwiegend gebirgig. Tätige und erloschene Einzelvulkane und Vulkangruppen bilden die zentrale Längsachse der Insel (mehrere Gipfel über 3 000m ü. M., Semeru 3 676m ü.M.). Beiderseits der Vulkanreihe (17 tätige Vulkane) gliedern sich Berg- und Hügelländer (aus tertiären Kalken und Mergeln) an. Im N[orden] sind dem Bergland z.T. versumpfte Schwemmlandebenen vorgelagert.
Aus: Brockhaus --- Die Enzyklopädie, Bd. 11, 20. Aufl., 1997, p. 149.
 - **Informatik:** objektorientierte, in starkem Maße an C++ angelehnte betriebssystem- und plattformunabhängige Programmiersprache, die 1995 von der Firma "Sun Microsystems" (USA) entwickelt wurde. J[ava] ist eine Sprache für in Netzen verteilbare Anwendungen, bei denen nicht nur Datensätze, sondern auch Programmteile (so genannte *Applets*;) in den Rechner geladen und ausgeführt werden können. Damit wird die in der Informatik, speziell im -> Internet, äußerst wichtige Forderung nach Systemunabhängigkeit erfüllt. J[ava]-Programme können auf beliebigen Rechnern ausgeführt werden, die über einen javafähigen WWW-Browser (-> World Wide Web) verfügen; zusätzlich können wie --- wie herkömml[iche] Programme --- auch ohne Browser ausgeführt werden, sofern das J[ava]-Laufzeitsystem direkt im Betriebssystem implementiert ist.
Aus: Brockhaus --- Die Enzyklopädie, Bd. 11, 20. Aufl., 1997, p. 149.
- Umgangssprachlicher Ausdruck für starken, heißen Kaffee in den USA
- In einem Satz (nach der Definition von [Sun Microsystems](#)): eine [einfache und kleine](#), [objektorientierte](#), [dezentrale](#), [interpretierte](#), [stabil laufende](#), [architekturneutrale](#), [portierbare](#) und [dynamische](#) Sprache, die [Hochgeschwindigkeitsanwendungen](#) und [Multithreading](#) unterstützt.

Im einzelnen bedeutet das:

- Java ist einfach; die Sprachkonzepte (Syntax) sind überschaubar. Im wesentlichen Stellen sie eine vereinfachte Untermenge der Programmiersprache C++ dar.
Wesentliche Unterschiede: stärker objektorientiert (d.h. keine *structs* mehr möglich, jedoch Beibehaltung des Konzepts der primitiven Werttypen), keine expliziten Zeiger (daher auch keine Zeigerarithmetik mehr möglich), keine Operatorüberladung, kein Präprozessor, keine Headerdateien (dafür Paketkonzept), automatische Speicherbereinigung durch Garbage Collection.
Klein zielt sowohl auf den Umfang der entstehenden ausführbaren Dateien, als auch die Größe der Java-Laufzeitumgebung ab. So werden Importdateien in Java nicht statisch inkludiert, sondern nur extern referenziert, wodurch die Größe teilweise signifikant verringert werden kann. Beispielsweise mißt das *HelloWorld*-Beispiel als Java-Bytecode 426 Byte, wohingegen das C-Pendant ([HelloWorld.c](#)) (gelinkt als command line executable für die Windows32-Plattform als [HelloWorld.exe](#)) über 34.000 Byte, bzw. unter Nutzung der Cygwin-Bibliothek mehr als 18.000 Byte, benötigt.
- Eine einzige Basisklasse *Object* als Wurzel der gesamten Klassenhierarchie. Keine Mehrfachvererbung (aber Nachbildung mit *Interfaces* möglich, wodurch die wesentlichen Nachteile eliminiert werden, jedoch die Vorteile gewahrt bleiben).
Mit Java-Beans steht ein eigenes Komponentenmodell zur Verfügung.
- Java erfüllt ein wesentliches Charakteristikum von Client/Server-Anwendungen, die Fähigkeit, Informationen und die Berechnungslast der Daten zu verteilen. Der Begriff *dezentral* beschreibt dabei die Beziehung zwischen Systemobjekten, gleichgültig ob diese Objekte auf lokalen oder entfernten Systemen abgelegt sind.
Am bekanntesten dürften die Java *Applets* sein, welche die Verteilung ganzer Applikationen über das World-Wide-Web (Abk. WWW).
- Mit Java erstellte Applikationen sind auf jeder Plattform (Hardware, Betriebssystem) ausführbar, für die eine Java Virtual Machine (Abk. JVM) existiert. Um dies zu erreichen wird durch den Java-Compiler kein plattformspezifischer Objektcode generiert, sondern eine *Zwischenrepräsentation* --- der *Byte-Code*. Dieser wird zur Ausführungszeit (auch: Laufzeit) durch das *Laufzeitsystem* --- die JVM --- *interpretiert* und für den Anwender transparent in maschinenspezifische Instruktionsfolgen umgesetzt. Eine JVM kann auch im *Web Browser* implementiert sein (z.B. [Netscape Navigator](#), [Microsoft Internet-Explorer](#)).
- Java ist *stabil* im Sinne von *zuverlässig*. Dies bedeutet in der Praxis die Verringerung der Absturzwahrscheinlichkeit (gleichbedeutend mit undefiniertem, unvorherseh- und -sagbarem Verhalten) eines mit der Sprache geschriebenen Programms.
Java ist (wie Pascal oder Oberon) eine streng typisierte Sprache, das bedeutet Typfehler können bereits zur Compilierzeit erkannt werden.
Laufzeitfehler können in Form von expliziten Ausnahmeroutinen behandelt werden (Exception Handling).
Durch den Verzicht auf explizite Zeiger wurde eine häufige Quelle für Laufzeit- und Speicherfehler eliminiert.
- Die Neutralität hinsichtlich der zugrundeliegenden Zielarchitektur (Hardware, Betriebssystem) wurde bereits bei der JVM angesprochen. Hier soll nochmals der Aspekt des architekturnabhängigen Bytecodes betont werden.
- Folgt (eigentlich) direkt aus der [Architekturneutralität](#); die Java-Laufzeitumgebung --- die virtuelle Maschine --- kann für beliebige Zielarchitekturen umgesetzt werden. Sofort danach sind alle Java-Applikationen, ohne Änderung, auf der neuen Plattform ablauffähig.
Das selbe gilt für mit Java erstellte Oberflächen, die ebenfalls nicht auf eine Plattform (Betriebssystem oder Window Manager) festgelegt sind.
Der --- bei nativ compilierenden Sprachen wie C/C++ notwendige --- Aufwand eine Applikation an eine neue Zielumgebung anzupassen (d.h. Portierung auf neue Hardware oder Betriebssystem (Zeigerlängen, Datentypgrößen, ...)) entfällt durch die Zwischenstufe der JVM, bzw. wird in die Umsetzung dieser auf die neuen Gegebenheiten verlagert.
- Die Architektur von Java ermöglicht es, die Sprache jederzeit an weiterentwickelnde Umgebungen anzupassen.
- Es versteht sich von selbst, dass plattformunabhängig ausgelagter Bytecode der zur Laufzeit jedesmal interpretiert und in plattformspezifischen Maschinencode umgesetzt werden muss, Geschwindigkeitsnachteile gegenüber auf genau eine Plattform zugeschnittenen Applikationen aufweist. Jedoch haben Untersuchungen von Sun gezeigt, dass Java-Bytecode --- der in plattformspezifischen Maschinencode übersetzt wurde --- plattformspezifischem Code anderer Sprachen ebenbürtig ist. (Hierdurch geht jedoch der Aspekt der Plattformunabhängigkeit verloren!).

Java ist jedoch reinen Scriptsprachen (wie Perl oder Python) an Ausführungsgeschwindigkeit signifikant überlegen. Im Allgemeinen ist ein Javaprogramm ca. 15-25 mal langsamer als optimierter C-Code. (Durch *Just in Time Compilation* läßt sich dieses Manko jedoch zumindest abfedern. Mittlerweile existieren auch plattformabhängige Compiler für Java, die nativen Binärcode erzeugen. Dieser erreicht üblicherweise die Geschwindigkeit von C++-Applikationen.)

- *Multithreading* soll (zunächst) vereinfacht als die Fähigkeit eines Programms definiert werden mehrere Tätigkeiten zur selben Zeit ausführen zu können.

Zur Synchronisation wird das Monitorkonzept nach Hoare direkt in der API realisiert.

▲ 1.2 Entstehungsgeschichte

- 1990: Auf Anraten des Programmierers Patrick Naughton wird bei [Sun Microsystems](#) eine Entwicklertruppe unter dem Namen *Green* ins Leben gerufen. Zentrale Mitarbeiter sind, neben Naughton, James Gosling und Mike Sheridan. Ziel des (anfänglich hochgeheimen) Projekts ist die Analyse des EDV-Marktes hinsichtlich verwertbarer Zukunftstrends.
Die Männer ermitteln als zukünftig erfolgversprechende Einnahmequelle für SUN den Bereich der Consumerelektronik (d.h. Telephone, Videorecorder, Wasch- und Kaffeemaschinen --- kurz allerlei elektronische Alltagsgeräte). Ferner definieren sie Anforderungen an ein zukünftiges plattformunabhängiges (d.h. universell einsetzbares) Betriebssystem dieser Maschinen. Vor allem Fehlertoleranz sowie leichte Bedienbarkeit und besondere Stabilität sollten dieses gegenüber existierenden Betriebssystemen auszeichnen. Die Softwaresteuerung sollte in C++ realisiert werden.
- Frühjahr 1991: erster Prototyp eines solchen Systems
- August 1991: Unter Federführung von James Gosling wird nach einer Sprache zur Programmierung des neu zu entwickelnden Betriebssystems gesucht; leider erfolglos.
Nicht das es im Sommer 1991 an möglichen Programmiersprachenaspiranten mangelte, sondern keiner der Kandidaten deckte das im Jahr zuvor formulierte Anforderungsspektrum zur Zufriedenheit ab.
Folglich wurde eine neue Sprache --- unter dem Namen *Oak* (engl. Eiche) --- entwickelt.
- 1992: Green-Team präsentiert ersten Prototyp ([Trickfigur Duke](#))
Die unabhängige Firma *FirstPerson* wird gegründet.
- März 1993: Neuorientierung des Projekts und Konzentration der Aktivitäten auf den Bereich interaktiven Fernsehens.
- April 1993: Der von Marc Andressen entwickelte erste graphische WWW-Browser NCSA Mosaic wird vorgestellt.
- Sept. 1993: Arthur van Hoff stößt zur Mannschaft von First Person.
- bis 1994 ist es FirstPerson nicht gelungen Oak zu vermarkten --- die Firma wird eingestellt, und das Produkt verschwindet in der Versenkung. Das Green-Team kehrt zu Sun zurück.
- 1994: Auf Initiative von William (Bill) Joy wird Oak für das Internet modifiziert (der mittlerweile legendäre *Mosaic*-Browser war ein Nebenprodukt der Arbeit des Green-Team).
- 1995: Oak wird im Internet zur Verfügung gestellt; allerdings nun unter dem neuen Namen *Java*. Vorstellung der β -Version auf Sun-World-Konferenz.
- 1996: Java Development Kit (Abk. JDK) v1.0
- 1997: JDK v1.1
- 1998: JDK v1.2
- 2000: JDK v1.3
- 2002: JDK: v1.4
- 2003: Erste Betaversion des JDK v1.5 (Codename *Tiger*)

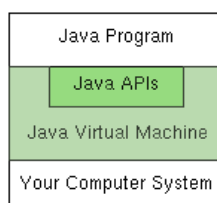


▲ 1.3 Die Java-Plattform

Prinzipiell kann, wie erwähnt, eine Java-Applikation auf jeder beliebigen Plattform zur Ausführung gebracht werden --- unabhängig davon welches Gerät die notwendige virtuelle Maschine implementiert.

So existieren Visionen und erste Umsetzungen von JVMs für Elektrogeräte wie Kaffeemaschinen, Kühlschränke, Radios etc.

Zur Erinnerung: die Intention der ursprünglichen Java-Entwicklung zielte nicht auf Desktop Rechner oder gar das Internet.



Die Abbildung schematisiert den Aufbau der Java-Plattform. Von der konkreten Plattform der physischen Hardware (Prozessor, etc. und des darauf ausgeführten Betriebssystems) wird durch die plattformspezifische virtuelle Maschine

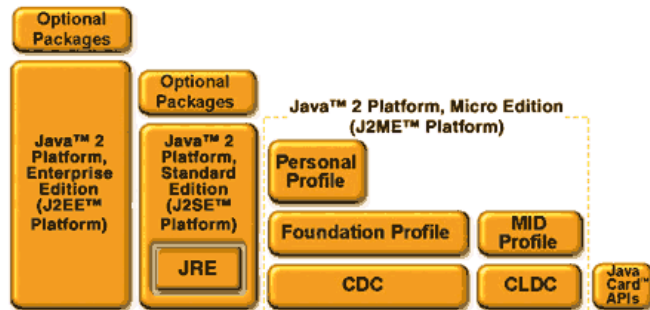
abstrahiert.

In diese eingebettet ist das *Java-Application Programmers Interface* (Abk. API) --- die sog. *Klassenbibliothek*. Sie ist vollständig in Java realisiert, damit ist sie bereits als plattformunabhängiges Java-Programm ausgelegt. Das API liefert die wesentlichen Grundprimitiven zur Erstellung leistungsfähiger Programme.

Auf dieser *virtuellen Plattform* läuft das (Benutzer) Java-Programm --- in Form des Bytecodes --- ab.

Hinweis: Beachten Sie die Unterscheidung zwischen Java-Plattform und Ausführungsplattform (=Betriebssystem und Hardware) einer Java-Applikation

Zur Applikationserstellung mit der Java-Plattform bietet SUN drei verschiedene Ausbaustufen (*Editionen*) an, die in der nachfolgenden Abbildung zusammengestellt sind.

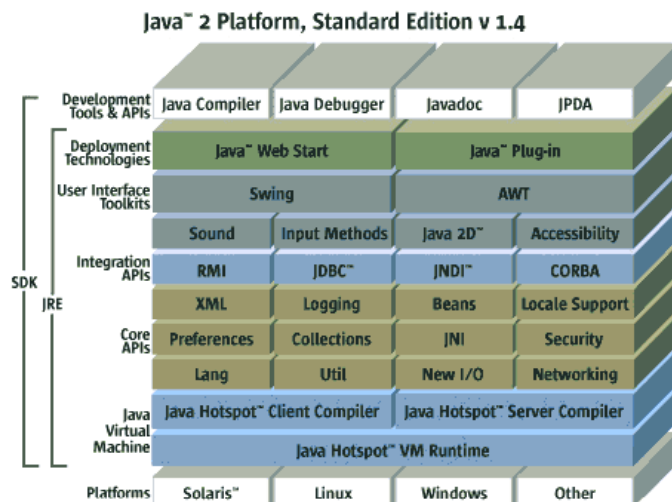


Diese Editionen basieren alle auf demselben Java-Sprachkern, verfügen jedoch über verschiedene Standard-APIs, die auf die jeweils adressierte Problemstellung zugeschnitten sind. Im Einzelnen sind dies:

- **Java 2 Enterprise Edition:** Erstellung von Webapplikationen, die durch einen Application Server ausgeführt werden.
- **Java 2 Standard Edition:** Erstellung von Clientapplikationen zur Ausführung auf Desktopmaschinen.
- **Java 2 Micro Edition:** Erstellung von Applikationen für ressourcenbeschränkte Geräte wie Mobiltelefone, PDAs, etc.

Wir werden uns nachfolgend auf die Betrachtung des Java-Sprachkernes, sowie ausgewählter APIs beschränken, überwiegend in allen drei Editionen, jedoch mindestens in J2EE und J2SE, zur Verfügung stehen.

Das Sun Java Development Kit (JDK)



Das JDK stellt eine Referenzimplementierung der kompletten Java-Plattform zur Verfügung. Seine Hauptbestandteile sind:

- **Java-Compiler:** `javac`
erzeugt Byte-Code
Minimalaufruf: `javac example.java`
Beispiel: `javac HelloWorld.java` erzeugt die Dateien `HelloWorld.class` und `SayHello.class`
- **Java-Interpreter:** `java`
Laufzeitumgebung zur Ausführung der Bytecode-Dateien
Minimalaufruf: `java example`
Wichtig: die Dateierweiterung `.class` darf nicht angegeben werden!
Beispiel: `java HelloWorld` führt die Applikation aus.
- **Applet-Viewer:** `appletviewer`
Dient zum Testen von Java-Applets ohne Web-Browser
Minimalaufruf: `appletviewer example.html`
Hinweis: Das Applet muß in einer Minimal-HTML-Seite referenziert werden, eine losgelöste Ausführung ist nicht möglich.
Beispiel (absolutes Minimalapplet (`HelloWorldApplet1.java`): `appletviewer HelloWorldApplet1.html`
- **Dokumentationswerkzeug:** `javadoc`
Erzeugt standardisierte HTML-Dokumentation aus Java-Quellcode und darin enthaltenen speziellen formalisierten Kommentaren.

Minimalaufruf: javadoc example.java

Beispiel: javadoc -private -version -author -windowtitle "Beispiel einer JavaDoc generierten Dokumentation" Hello World.java

Generiert verschiedenste Dokumentationsdateien im HTML- bzw. CSS-Format. Als Einstiegspunkt empfiehlt sich die Datei [index.html](#)

- Textmode-Debugger: jdb
Einfacher textorientierter Debugger
Minimalaufruf: jdb example
- Disassembler: javap
Rückübersetzer zur Wiedergewinnung von Java-Quellcode aus Bytecode-Dateien
Minimalaufruf: javap example

Die Rückübersetzung des Compilates [HelloWorld.class](#) mit javap -c HelloWorld liefert:

```
(1)Compiled from HelloWorld.java
(2)public class HelloWorld extends java.lang.Object {
(3)    public HelloWorld();
(4)    public static void main(java.lang.String[]);
(5)}
(6)
(7)Method HelloWorld()
(8)    0 aload_0
(9)    1 invokespecial #1 <Method java.lang.Object()>;
(10)   4 return
(11)
(12)Method void main(java.lang.String[])
(13)   0 getstatic #2 <Field java.io.PrintStream out>;
(14)   3 ldc #3 <String "Hello World!">;
(15)   5 invokevirtual #4 <Method void println(java.lang.String)>;
(16)   8 return
```

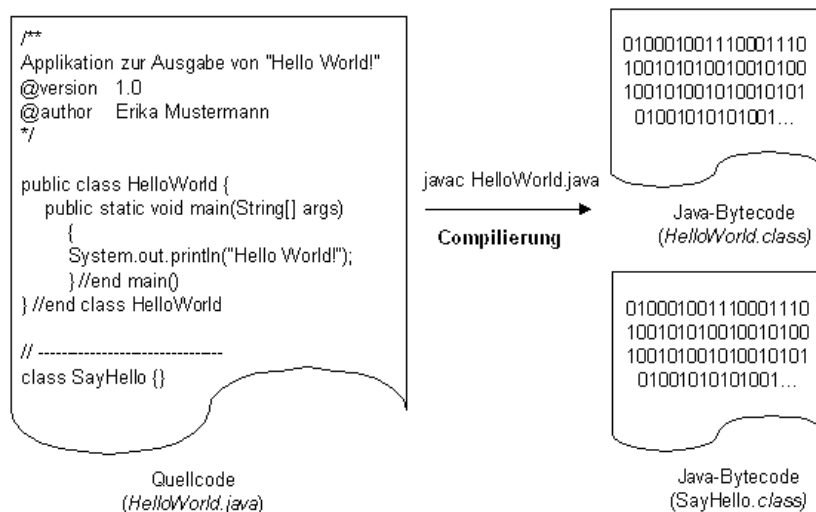
Beispiel 1: Decompilierung mit javap

Anmerkung: Zum Investitionsschutz kommerziell erstellter Java-Software sind Applikationen verfügbar, die den generierten Bytecode so nachbearbeiten, daß eine Decompilierung deutlich erschwert wird, oder die Ausgabe unbrauchbar wird. Die Ausführbarkeit des Bytecodes wird hierdurch nicht beeinträchtigt.

Zusätzlich ist ein eigenständiger Java-Interpreter, das sog. *Java Runtime Environment* (Abk. JRE) verfügbar. Sein Minimalaufruf lautet jre example. Diese Applikation ist nicht Bestandteil der Standard Java-Plattform (JDK v1.3) und ist separat kostenlos über die Sun Java-Homepage beziehbar.

▲ 1.4 Vom Quellcode zum lauffähigen Programm

Die Programmentwicklung vollzieht sich im klassischen edit-build-run-Zyklus. Als Besonderheit generiert der Java-Compiler je eine Bytecode-Datei (Dateiextension class) für jede zugreifbare Klasse innerhalb der Quelldatei. Die entstehenden Class-Dateien werden durch die Laufzeitumgebung interpretativ ausgeführt.



Im Beispiel enthält die Quellcodedatei [HelloWorld.java](#) die beiden Klassendateien [HelloWorld](#) und [SayHello](#). Diejenige Klasse, welche die Main-Methode beinhaltet muß identisch der sie enthaltenden Quelldatei benannt sein -- im Beispiel HelloWorld.

Mit javac HelloWorld.java wird die Datei übersetzt, und die beiden Class-Dateien HelloWorld.class und SayHello.class erzeugt.

Die Programmausführung erfolgt durch Absetzen von java HelloWorld auf der Kommandozeile. (Achtung! Keine Dateiextension angeben!)

Der Aufruf `java SayHello` führt hingegen wegen des Fehlens der `Main`-Methode in der Klasse `SayHello` zur Fehlermeldung `Exception in thread "main" java.lang.NoSuchMethodError: main`.

▲ 2. Syntax und Semantik der Programmiersprache Java

▲ 2.1 C, C++, C# und Java

Wie bereits in der [Einführung angedeutet](#) wurde Java nahe an der Syntax der verbreiteten hybrid objektorientiert-prozeduralen Sprache C++ entwickelt. Jedoch mit der Einschränkung, deren mitunter kryptischen Syntax deutlich zu vereinfachen. Überdies wurde auf wesentliche dort anzutreffende Sprachmerkmale verzichtet.

Die [Java Language Specification](#) führt bereits im Vorwort aus, daß sich die Java-Sprachentwickler zwar in einer Vielzahl von Punkten an C und C++ orientierten, jedoch der Sprachaufbau stark von diesen Beiden Vätern abweicht. Aus praktischer Motivation heraus wurde beim Sprachdesign auf die Einführung neuer und ungetesteter Sprachelemente verzichtet.

Wie [erwähnt](#) ist Java als streng typisierte Sprache ausgelegt. Daher können die meisten Typfehler bereits zur Übersetzungszeit erkannt werden. Technisch gesehen meint strenge Typisierung (auch *statische* Typisierung), daß der Typ einer Variable während der Programmausführung nicht verändert werden kann. Durch statische Typanalyse während des Compilierungsvorganges können Typfehler erkannt werden. Durch polymorphe Aufrufe kann es jedoch auch während der Programmausführung zu Typfehlern kommen, da hierbei der dynamische Typbindung zum Einsatz kommt.

Anders als C und C++ verfügt Java über keinerlei systemnahe Konstrukte. Direkte Hardwarezugriffe und Systemprogrammierung im Allgemeinen kann daher mit nicht realisiert werden. Die Sprachväter begründen dies mit der Transparenz einer *high-level*-Sprache, die keinerlei Rückschlüsse auf die darunterliegende Hardware zulassen sollte.

Anders als C/C++ verfügt Java über [automatische Speicherbereinigung](#) (engl. *garbage collection*), welche die fehlerträchtigen Speicheroperationen (insbesondnere *free* und *delete*) obsolet werden läßt. Konsequenterweise läßt Java die Allokation beliebiger Speicherbereiche nicht zu. Nicht mehr benötigte Speicherplätze (Variablen) können zwar durch den Programmierer (durch die explizite Zuweisung von NULL) als nicht mehr benötigt gekennzeichnet werden, Freigabefunktionen stellt die Sprache jedoch nicht zur Verfügung.

Bekannte, und berühmte, Fehlerquellen wie Arrayzugriffe ohne vorherige Indexprüfung, variabel lange Parameterlisten oder Zeiger nebst Zeigerarithmetik existieren nicht, und verleihen dem Sprachentwurf dadurch zusätzliche Sicherheit.

Der Verzicht auf (explizite) Zeiger zieht die Behandlung aller Methodenparameter als Werte (*call-by-value*) nach sich. Ebenso wurde auf Umgebungseigenschaften wie den Präprozessor und Includedateien vollständig verzichtet wurde. Die Strukturierung des Java-Quellcodes kann über ein eigenes Paketkonzept erfolgen.

Nützlichkeiten wie die Lockerung des Variablendefinitionszwanges am Blockanfang wurden beibehalten. Selbiges gilt für die Aufhebung der Trennung zwischen Deklaration und Definition bei einfachen Variablen, wie sie bereits in anderen Sprachen verwirklicht ist. Für Objekte existiert diese Trennung -- sinnvollerweise -- weiterhin fort.

Hinsichtlich objektorientierter Konzepte geht Java deutlich über C++ hinaus. Dergestalt, daß weder klassenlos Programme entwickelt werden können, noch Structs und Unions als Zwitter zwischen Variablen und Klassen implementiert sind. Die Prämisse strengerer Objektorientierung erklärt auch das Verbot globaler Variablen und Methoden. Allerdings sind die primitiven skalaren Datentypen (wie `int`, `char`, etc.) nicht als Objekte realisiert. Analog C++ können Methodennamen überladen werden (*Ad-hoc Polymorphie*). Zur Eindeutigkeitsidentifikation wird die Signatur (gebildet aus Methodennamen und übergabeparametern) herangezogen. Die Überladung von Operatoren, wie in C++ möglich, ist nicht vorgesehen.

Der in C++ realisierte Vererbungsmechanismus wurde unter der Einschränkung übernommen, Mehrfachvererbung zu verbieten. Um trotz dieser Restriktion sinnvolle Anwendungsentwicklung betreiben zu können wurde Java als zusätzliches Sprach-Konzept die Schnittstelle (engl. *Interface*) hinzugefügt. Hiervon können eine Klasse beliebig viele implementiert werden.

Im Gegensatz zu C++ erben Klassen ohne ausdrücklich angegebene Elternklasse automatisch per Vorgabe von [java.lang.Object](#).

Parametrische Polymorphie in Form von Templates, wie in C++ anzutreffen, ist in Java erst ab der Sprachversion 1.5 realisiert.

Von C++ wurde die Fehlerbehandlung in Form von Ausnahmen (engl. *exceptions*) übernommen. Hierdurch können Fehlerereignisse zentralisiert behandelt werden. Zusätzlich wird der Kontrollfluß von Fehlerbehandlungscode bereinigt und dadurch übersichtlicher.

Offensichtlich ist die Übernahme des Kommentierungsstils von C und C++. So können alle Kommentarkonstrukte wie in den vorgenannten Sprachen üblich eingesetzt werden.

Zusätzlich wird ein separater Kommentarstil (`/**...*/`) für Quellen automatisierter Dokumentation angeboten.

Zwischen den so abgegrenzten Regionen können vorgegebene Marken plziert werden; diese werden beispielsweise durch das JDK-Werkzeug [javadoc](#) verarbeitet.

Augenfälligstes abgrenzendes Merkmal von Java gegenüber C/C++ ist es, daß kein plattformspezifischer Maschinencode als Resultat des Compilierungsvorganges erzeugt wird, sondern binärer Bytecode genannte Zwischenrepräsentation. Diese wird durch die Java Virtual Machine zur Ausführungszeit interpretativ abgearbeitet. Anmerkung: Bytecode ist nicht Java-spezifisch, sondern kann auch durch andere Sprachen erzeugt werden, was mitunter auch geschieht.

▲ Microsofts C#

Die durch Microsoft entwickelte Sprache C# (sprich: *C sharp*) weist einige interessante Parallelen zu Java auf, führt jedoch auch neue Konzepte ein.

Jenseits der Einordnung in die Komponentenarchitektur der .NET-Plattform, welche nur schwer mit der der Java-Plattform (insbesondere der J2EE-Plattform) verglichen werden kann, stellt C# jedoch eine vollwertige -- sehr stark an Java orientierte -- Programmiersprache dar. Bereits das, zu erwartende, Standardbeispiel der HelloWorld-Applikation läßt die enge Verwandtschaft, abzulesen an der Syntax deutlich werden (Vergleiche HelloWorld [in Java](#), in [in C#](#)).

Im Gegensatz zu C# ist Java auf der Programmierenebene nicht *rein* objektorientiert wie beispielsweise SmallTalk. So treten auch hier die primitiven Datentypen als nicht-objektartige Werte auf.

Ebenso wie in Java ist ausschließlich Einfachvererbung zugelassen, ergänzt wird diese um Schnittstellen. Auch C# erlaubt es einer Klasse mehrere Schnittstellen zu implementieren. Die Syntax unterscheidet jedoch nicht mehr explizit zwischen Schnittstellenimplementierung und Erben von einer Klasse (siehe [Beispiel](#)). Unverändert zu Java wird auch jede Klasse, die über keine Elternklasse verfügt automatisch als Subklasse von `object` eingeordnet. Blattknoten einer Vererbungshierarchie (d.h. Klassen für die durch den Programmierer vorgegeben wird, sie sollen nicht weiter durch Ableitung spezialisiert werden) können mit dem Schlüsselwort `sealed` -- in Java: `final` -- gekennzeichnet werden.

Zusätzlich zu Klassen beinhaltet C#, wieder, den Sprachmechanismus der Struktur (`struct`). Hierüber wird die Differenzierung zwischen Wert- und Referenztypen realisiert. Während Klassen, Java-konform Referenztypen bezeichnen, übernehmen Strukturen die der Werttypen. Desweiteren werden Structs durch den Compiler als Bestandteile umgebender Objekte übersetzt. Durch diese Charakteristika unterscheiden sich C#-Structs stark von den dortigen Klassenstrukturen, weshalb sie auch nicht analog zu deren Schema verwirklicht sind. Strukturen können weder erben noch vererbt werden, lediglich die Möglichkeit Schnittstellen zu implementieren bleibt erhalten.

Deutlich komfortabler als in Java fällt jedoch der Umgang mit Referenz- und Werttypen aus. Das in C# angebotene *boxing* und *unboxing* übertrifft deutlich die Benutzungsfreundlichkeit der Java-Wrapperklassen ([Beispiel](#)). Dieser Mißstand wurde jedoch in der Javasprachversion 1.5 behoben, die dieses Konzept ebenfalls einführt.

Erstmals reicht C# auch den Sprachumfang C/C++-basierter Sprachen um neue syntaktische Konstrukte an. Hierunter fallen beispielsweise Schlüsselwörter zur Definition von Delegationsobjekten, die als solche explizit im Programmcode deklariert werden können.

Wie Java compiliert auch C# in eine Zwischenrepräsentation. Allerdings mit dem Unterschied, daß diese nicht interpretativ abgearbeitet wird, sondern erneut in maschinenspezifisches natives Format übersetzt wird.

Die Tabelle stellt einige Sprachmerkmale von C++, Java und C# gegenüber

(Tabelle in Anlehnung an: Eisenacker, U.: Dissonanz oder Wohlklang, in: iX 9/2000, Hannover, 2000, p. 48-51)

Sprachmerkmal	C++	Java	C#
Automatische Speicherbereinigung (garbage collection)			
Coercion (Typumwandlungspolymorphie)			
Globale Methoden			
Inklusionspolymorphie			
Operatoroverloading			
Parametrische Polymorphie (Templates)		 (1)	 (2)
Referenztypen			
Überladungspolymorphie			
Werttypen			 <i>unified type system</i>



- (1) Ab Version 1.5 für Klassen der Collection API im Sprachumfang enthalten.
 (2) Die Aufnahme in die Sprache ist für die Nachfolgeversion des .NET-Frameworks 1.1 geplant.
 Einen Ausblick auf die geplanten Sprachmerkmale liefert das Projekt [CLGEN](#) von Microsoft Research.

▲ 2.2 Grundstrukturen

▲ 2.2.1 Programmaufbau

Bereits am [HelloWorld-Beispiel](#) wird der wesentliche Aufbau einer Javaquellcodedatei sichtbar.

```
(1)public class Minimal {
(2)    public static void main(String[] args) {
(3)        //..
(4)    } //main()
(5)} //class Minimal
```

Beispiel 2: Minimales Java-Programm

Zunächst erfolgt die Definition einer Klasse; im Beispiel `Minimal`.

Vor dem Schlüsselwort `class` ist die Sichtbarkeit auf `public` festgelegt, was eine allgemeine Sichtbar- und Zugriffsbarkeit der Klasse bewirkt.

Im Gegensatz zu C++ ist jedoch die Klassendefinition zwingend erforderlich! Es ist nicht möglich Javaquellcode, der keine Klassendefinition enthält, zu übersetzen. Hingegen ist es ohne weiteres möglich gewöhnliche C-Programme mit einem C++-Compiler zu übersetzen.

Darüberhinaus ist es zwingend vorgeschrieben die Quellcodedatei identisch zur `public` deklarierten Klasse zu benennen.

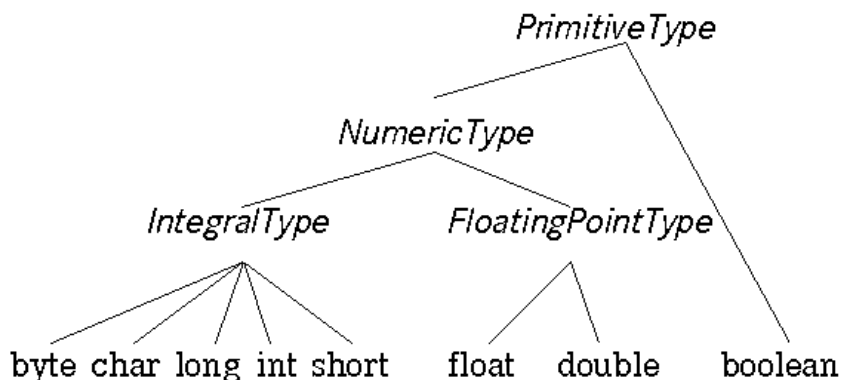
Eine weitere Besonderheit gegenüber C++ stellt die Platzierung der `main`-Funktion dar. Während diese in C++, selbst bei der Verwendung von Klassen ([siehe Beispiel \(HelloWorld.cpp\)](#)), außerhalb jeder Klasse angeschrieben werden muß, um automatisch beim Programmstart ausgeführt zu werden, erzwingt Java die `main`-Methode innerhalb einer `public` deklarierten Klasse.

Die Signatur der `main`-Methode ist mit `public static void main(String[] args)` fixiert. Davon Abweichende Spezifikationen sowohl in Rückgabewert als auch Parameterliste, können zwar in Bytecode übersetzt werden, jedoch wird diese abweichende `Main`-Methode nicht mehr automatisch durch das Laufzeitsystem aufgerufen. Auch hier zeigt sich Java deutlich restriktiver als C/C++, die beide beliebige Rückgabetypen und -- in Grenzen -- leicht variierende Parameterlisten zulassen.

Abschließend: Innerhalb der `main`-Methode der `public` deklarierten Klasse kann der auszuführende Code angegeben werden.

▲ 2.2.2 Einfache Datentypen

Wie herkömmliche prozedurale Programmiersprachen auch stellt Java primitive Datentypen zur Verfügung. Im Gegensatz zu manchen etablierten objektorientierten Programmiersprachen (z.B. SmallTalk) sind diese Datentypen in Java intern nicht als Objekte realisiert.



Im Einzelnen werden angeboten:

Datentyp Erklärung	Wertebereich
<code>boolean</code> Wahrheitswert	true oder false
<code>char</code> Einfaches Zeichen aus dem 16-Bit Unicode Zeichensatz	
<code>byte</code> 8-bittige vorzeichenbehaftete Ganzzahl	-2 ⁷ bis 2 ⁷ -1 -128 <= byte <= 127
<code>short</code> 16-bittige vorzeichenbehaftete Ganzzahl	-2 ¹⁵ bis 2 ¹⁵ -1 -32768 <= short <= 32767
<code>int</code> 32-bittige vorzeichenbehaftete Ganzzahl	-2 ³¹ bis 2 ³¹ -1 -2147483648 <= int <= 2147483647
<code>long</code> 64-bittige vorzeichenbehaftete Ganzzahl	-2 ⁶³ bis 2 ⁶³ -1 -9223372036854775808 <= long <= 9223372036854775807
<code>float</code> 32-bittige Gleitkommazahl (nach IEEE 754-1985)	Größtmögliche positive Zahl: 3.40282347e+38f Kleinstmögliche positive Zahl: 1.40239846e-45f
<code>double</code> 64-bittige Gleitkommazahl (nach IEEE 754-1985)	Größtmögliche positive Zahl: 1.79769313486231570e+308 Kleinstmögliche positive Zahl: 4.94065645841246544e-324
<code>Object</code> Objektwertiger Datentyp. Jedes Objekt hat (implizit) diesen Typ.	

Siehe [Java Language Specification](#)

Anmerkungen:

- Der Wahrheitswert ist nicht wie in C++ üblich als Ganzzahl realisiert. Auch eine explizite Typumwandlung ist nicht möglich, und wird zur Übersetzungszeit (als Typfehler) abgefangen (Compilerfehlermeldung: `inconvertible types`). Hinweis: Zwar ist im ANSI-C++-Standard ein `bool` als expliziter Datentyp (üblicherweise der Länge 1 Byte) definiert, der jedoch die Wahrheitswerte als Integer-Zahlen ablegt. So gilt: `true == 1` bzw. `false == 0` ([siehe Beispiel](#)).
- Die Ganzzahldarstellung erfolgt immer im Zweierkomplement, weshalb der positive Wertebereich eine Zahl weniger als der negative umfaßt.
- Alle numerischen Typen sind vorzeichenbehaftet. Vorzeichenlose Pendant (wie in C durch Voranstellen eines *unsigned* bildbar) existieren nicht.
- Die Bitlänge aller Datentypen ist festgelegt, und nicht plattformabhängig variierbar.
- Für die beiden Gleitkommatypen existiert ein Literal zur Darstellung von Plus- und Minus-Unendlich. Zusätzlich ein mit NaN (*Not a Number*) ein ausgezeichnete Wert für nicht interpretierbare Belegungen.
- `char` ist zwar als Zeichensymbol ausgelegt, kann aber auch in mathematischen Termen anstelle von `short` verwendet werden; hier wird `char` jedoch vorzeichenlos(!) interpretiert. Die Konversion von `char` zu `int` erfolgt implizit, die Umgekehrung bedarf jedoch einer ausdrücklichen Typumwandlung.
- Die nicht standardkonforme Implementierung der virtuellen Maschine von Microsoft bietet mit `object` (32-bittige Referenz auf ein Java-Objekt) und `returnAddress` (32-Bit) zwei zusätzliche Datentypen an.
- Ebenso wie in C/C++ üblich können Konstanten durch spezielle Muster typisiert werden.
 - `\uxxxx` für Unicodezeichen (`xxxx` ist die hexadezimale Zeichennummer)
 - `...l` oder `...L` für Longwerte
 - `0...` für Oktalwerte
 - `0x...` für Hexadezimalwerte
 - `...f` oder `...F` für Floatwerte
 - `...d` oder `...D` für Doublewerte
- Für die primitiven built-in Typen wird keine über- oder Unterlauferkennung angeboten. ([Siehe Beispiel Overflow.java](#)).
- Auffallend ist, daß keine Arraytypen existieren. Diese sind in Java nicht als Aggregation primitiver built-in Typen realisiert, sondern als Klasse der API umgesetzt.

```
(1)public class ConstFormats {
(2)    public static void main(String[] args) {
(3)        int i = 052;           //octal
(4)        System.out.println("i as decimal = "+i);
(5)
(6)        i = 0x2a;              //hexadecimal
(7)        System.out.println("i as decimal = "+i);
(8)
(9)        long l = 0x2aL; //hexadecimal long
(10)       System.out.println("l as decimal = "+l);
(11)
(12)       float f = 0x2af;       //hexadecimal float
(13)       System.out.println("f as decimal = "+f);
(14)    } //main()
(15)} //class ConstFormats
```

Beispiel 3: Verschiedene Deklarationen und Wertebelegungen [ConstFormats.java](#)

Das Programm liefert folgende Ausgabe:

```
i as decimal = 42
i as decimal = 42
l as decimal = 42
f as decimal = 687.0
```

VORSICHT! Die hexadezimale Definition des Floatwertes liefert nicht -- wie vielleicht intuitiv zu erwarten -- 42 als ganzzahligen Anteil, sondern die Gleitkommainterpretation des hexadezimal spezifizierten Bitmusters.

Als streng typisierte Sprache muß jede Variable in Java mit einem Typ deklariert werden, ungetypte Variablen existieren nicht.

Nach der Deklaration in einem Programm wird der Variableninhalt auf einen reservierten, für den Programmierer nicht zugänglichen Wert *undefined* gesetzt. Hierdurch kann der Compiler lesende Referenzierungen vor Wertdefinition zur Übersetzungszeit erkennen.

Die Deklaration geschieht, angelehnt an C/C++ durch Angabe des Typs gefolgt durch den Variablennamen. Optional kann dieses Statement durch die Festlegung eines Vorgabewertes ergänzt werden.

```
(1)public class NotInitialized {
(2)    public static void main (String[] args) {
(3)        int i;
(4)        System.out.println("i= "+i); //i is not initialized yet
(5)    } //main()
(6)} //class notInitialized
```

Beispiel 4: Nicht initialisierte Referenzierung [NotInitialized.java](#)

Liefert beim Übersetzen die Fehlermeldung:

```
NotInitialized.java:6: variable i might not have been initialized
    System.out.println("i= "+i); //i is not initialized yet
                        ^
1 error
```

```
(1)public class CharArithmetic {
(2)    public static void main (String[] args) {
(3)        char c = 'a';
(4)
(5)        System.out.println("c = "+c);
(6)        c++;
(7)        System.out.println("c = "+c);
(8)
(9)        int i = c;
(10)
(11)        System.out.println("i = "+i);
(12)        System.out.println("i = "+ (char) i);
(13)    } //main()
(14)} //class CharArithmetic
```

Beispiel 5: Verwendung von char als numerischer Typ [CharArithmetic.java](#)

Das Programm liefert bei der Ausführung folgende Ausgabe:

```
c = a
c = b
i = 98
i = b
```

▲ 2.2.3 Operatoren

Java bietet 37 verschiedene Operatoren an, die ihrer Semantik nach mit denen von C/C++ weitestgehend identisch sind ([siehe Language Specification](#)).

Je nach Typ auf dem Operator definiert ist, wird unterschieden zwischen: Integralen-, Fließkomma-, Boole'schen- und objektwertigen-Operatoren.

▲ Operatoren auf integralen Typen (

siehe Java API Specification)

- Vergleichsoperatoren, die als Resultat boolean liefern:
 - numerischer Vergleich: <, <=, > und >= ([siehe Java Language Specification](#))
 - numerische Gleichheit: == und != ([siehe Java Language Specification](#))
- Numerische Operatoren, die ganzzahligen Wert (int oder long) zurückliefern:
 - unäres Plus und Minus ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).
 - multiplikative Operatoren *, / und % ([siehe Java Language Specification](#)).
 - additive Operatoren + und - ([siehe Java Language Specification](#)).
 - inkrement Operatoren (Prefix- und Postfixvariante) ++ ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).
 - dekrement Operatoren (Prefix- und Postfixvariante) -- ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).
 - Shift Operatoren <<, >> bzw. >>> (Einführung von Füllnullen) ([siehe Java Language Specification](#)).
So gilt: -2 >>> 2 = -1 aber: -2 >> = 1073741823
 - bitweises Komplement ~ ([siehe Java Language Specification](#)).
 - bitweise Integeroperatoren &, | und ^ ([siehe Java Language Specification](#)).
- Konditionaloperator (Kurzschreibweise für if) ? : ([siehe Java Language Specification](#)).
- explizite Typumwandlung (cast), zur Umwandlung jedes Integraltypen in einen beliebigen numerischen ([siehe Java Language Specification](#), [siehe Java Language Specification](#)).

Diese built-in Operatoren nehmen keinerlei Fehlerprüfung oder -meldung vor. Lediglich die beiden Integerdivisionsoperatoren / und % werfen im Fehlerfall eine ArithmeticException-Ausnahme falls der Divisor gleich Null ist ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).

▲ Operatoren auf Fließkommatypen (

siehe Java Language Specification)

Auf diesen Typen sind die auch auf integralen Typen zugelassenen arithmetischen-, numerischen Vergleichs-, Inkrement- und Dekrement-Operatoren verfügbar. Ferner existiert der numerische Cast ([siehe Java API Specification](#)).

Auch die Fließkommaoperatoren lösen keine Exceptions aus. überlauf wird als positiv Unendlich, Unterlauf entsprechend als negativ Unendlich dargestellt. Liefert die Operation kein mathematisch interpretierbares Resultat, wird der Wert auf NaN gesetzt. In der Konsequenz resultiert auch NaN falls ein so gesetzter Operand erneut verknüpft wird. Dies gilt nicht für die Gleichheitsoperation.

- Vergleichsoperatoren, die als Resultat boolean liefern:
 - numerischer Vergleich: <, <=, > und >= ([siehe Java Language Specification](#))
 - numerische Gleichheit: == und != ([siehe Java Language Specification](#))
- Numerische Operatoren, die ganzzahligen Wert (float oder double) zurückliefern:
 - unäres Plus und Minus ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).
 - multiplikative Operatoren *, / und % ([siehe Java Language Specification](#)).
 - additive Operatoren + und - ([siehe Java Language Specification](#)).
 - inkrement Operatoren (Prefix- und Postfixvariante) ++ ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).
 - dekrement Operatoren (Prefix- und Postfixvariante) -- ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).
- Konditionaloperator (Kurzschreibweise für if) ? : ([siehe Java Language Specification](#)).
- explizite Typumwandlung (cast), zur Umwandlung jedes Integraltypen in einen beliebigen numerischen ([siehe Java Language Specification](#), [siehe Java Language Specification](#)).

Anmerkung: Identisch zu ANSI C/C++ wurde der unäre Operator + lediglich aus Symmetriegründen zum unären eingeführt.

▲ Operationen auf Boole'schen Typen

Auf Boole'schen Typen sind alle relationalen und logischen Operatoren zugelassen.

Als Operand des Konditionaloperators können neben Boole'schen Werten auch Integralwerte angegeben werden. Diese werden gemäß C-Konvention zu true konvertiert sofern sie ungleich Null sind, andernfalls zu false.

- relationale Operatoren (Gleichheit und Ungleichheit) == und != ([siehe Java Language Specification](#)).
- logisches Komplement ! ([siehe Java Language Specification](#)).
- logische Operatoren &, ^ und | ([siehe Java Language Specification](#)).
- konditionelles-und und konditionelles-oder && bzw. || ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).

Die Auswertung erfolgt im sog. short circuit-Verfahren. Hierbei wird nicht zwingend der gesamte Ausdruck ausgewertet. Bei der *Ver-und-ung* wird abgebrochen, sobald ein falsches Ergebnis vorliegt; bzw. bei der *Ver-oder-ung* sobald ein wahres Ergebnis vorliegt.

- Konditionaloperator `?` : ([siehe Java Language Specification](#))

▲ Operatoren auf objektwertigen Typen

- `+` zur Stringkonkatenation. Ist einer der beiden angegebenen Operanden nicht vom Typ `String`, so wird zur Laufzeit eine Stringkonversion durchgeführt ([siehe Java Language Specification](#)). Diese Stringkonversion ist immer möglich, da alle Objekte von `java.lang.Object` die Methode `toString` erben.
Anmerkung: hierbei handelt es sich strenggenommen um einen überladenen Operator
- Abfragen auf den Typ eines Objekts können durch den `instanceof`-Operator erledigt werden. Er liefert einen Wahrheitswert zurück.
Syntax: `object instanceof class`.

▲ Schlussbemerkungen

- Wie in C/C++ auch existieren die abkürzenden Zuweisungsschreibweisen: `+=`, `-=`, `*=`, `/=`, `&=`, `|=`, `^=`, `%=`, `<<=`, `>>=`, `>>>=`.
- Die aus C/C++ bekannten Speicherzellen-bezogenen Operatoren `*` und `&` werden wegen des Fehlens expliziter Speicherreferenzen nicht unterstützt.
- Ebenso der `sizeof`-Operator. Für ihn existiert kein Anwendungsfall, da einerseits die Größen der built-in Typen festgelegt ist, andererseits keine Speicherblöcke wahlfrei allokiert werden können.
- Die Operanden aller diskutierten Operatoren, außer `&&`, `||` und `?:`, werden vor der Ausführung des Operators ausgewertet.
Dies gilt insbesondere für Operatoren die eine Exception auslösen können. Eine solche wird ggf. nach Auswertung aller Operanden erzeugt.

▲ Operatorpräzedenz

Es gelten folgende Operatorpräzedenzen (geordnet von oben (entspricht höchster Präzedenz) nach unten (niedrigster Präzedenz)):

Operator	Symbol
Postfix-Operatoren	<code>[] . (params) expr++ expr--</code>
unäre Operatoren	<code>++expr --expr +expr -expr ~ !</code>
Erzeugung oder Typumwandlung	<code>new (type)expr</code>
Multiplikationsoperatoren	<code>* / %</code>
Additionsoperatoren	<code>+ -</code>
Verschiebeoperatoren	<code><< >> >>></code>
Vergleichsoperatoren	<code>< > <= >= instanceof</code>
Gleichheitsoperatoren	<code>== !=</code>
Bitoperator Und	<code>&</code>
Bitoperator exklusives Oder	<code>^</code>
Bitoperator inklusives Oder	<code> </code>
logisches Und	<code>&&</code>
logisches Oder	<code> </code>
Konditionaloperator	<code>?:</code>
Zuweisungsoperatoren	<code>= += -= *= /= %= >>= <<= >>>= &= ^= =</code>

▲ Automatische Typkonversion

Die automatische Typkonversion wird durch den Compiler bzw. das Laufzeitsystem immer dann angewandt, wenn nicht alle Operanden typgleich sind. Dies ist beispielsweise bei der Multiplikation einer `int`-Zahl mit einem Fließkommawert der Fall.
Die automatische Typumwandlung ist im Wesentlichen identisch zur in C/C++ realisierten ausgelegt.

	byte	char	short	int	long	float	double	boolean
byte	int	int	int	int	long	float	double	
char		int	int	int	long	float	double	
short			int	int	long	float	double	
int				int	long	float	double	
long					long	float	double	
float						float	double	
double							double	
boolean								boolean

Die Tabelle stellt die automatischen Typkonversionen zur Festlegung des Ausdruckstyps dar. Da alle Operatoren kommutativ sind, ist die Tabelle symmetrisch zur Hauptdiagonale (nur die obere Dreiecksmatrix ist aus übersichtlichkeitsgründen ausgefüllt).

Alle Typkonversionen werden statisch und operatorunabhängig durchgeführt, d.h. mögliche Typfehler werden zur Compilezeit erkannt und gemeldet.

▲ 2.3 Kontrollstrukturen

▲ 2.3.1 Selektion und Mehrfachselektion -- das if und case-Statement

Das if-Statement ist in Java analog der aus C/C++ bekannten Semantik und Syntax realisiert ([Siehe Language Specification](#)):

IfThenStatement:
if (Expression) Statement

IfThenElseStatement:
if (Expression) StatementNoShortIf else Statement

IfThenElseStatementNoShortIf:
if (Expression) StatementNoShortIf else StatementNoShortIf

```
(1)public class IfTest {
(2)    public static void main(String[] args) {
(3)        int i = 42;
(4)
(5)        if (1==1)
(6)            if (i < 50)
(7)                i++;
(8)            else
(9)                i--;
(10)    } //main()
(11)} //end class IfTest
```

Beispiel 6: If-Statement mit dangling else [IfTest.java](#)

Leider wurde in Java die Chance zur Ausmerzung des lästigen und fehlerträchtigen *dangling else-Problems* nicht genutzt. Daher wird -- konform zu C/C++ -- ein *else* immer dem letzten vorhergehenden *if* zugeordnet. Strenger im Vergleich zu C/C++ ist hingegen die Typisierung der Expression als Bedingung innerhalb der Verzweigung gefaßt. Hier ist zwingend der Typ boolean erforderlich.

Auch das switch-Statement ist identisch zum C/C++-Analogon realisiert. Ebenso wie dort müssen die Alternativzweige mit einem expliziten *break* abgeschlossen werden. Innerhalb der case-Verzweigungen sind nur konstante Ausdrücke der Typen char, byte, short oder int zugelassen. ([siehe Java Language Specification](#)).

Syntax:

```
SwitchStatement:
    switch ( Expression ) SwitchBlock

SwitchBlock:
    { SwitchBlockStatementGroupsoptional SwitchLabelsoptional }

SwitchBlockStatementGroups:
    SwitchBlockStatementGroup
    SwitchBlockStatementGroups SwitchBlockStatementGroup

SwitchBlockStatementGroup:
    SwitchLabels BlockStatements

SwitchLabels:
    SwitchLabel
    SwitchLabels SwitchLabel

SwitchLabel:
    case ConstantExpression :
    default :
```

Beispiel eines switch-Statements:

```
(1) public class SwitchTest {
(2)     public static void main(String[] args) {
(3)         int i = 42;
(4)
(5)         here:
(6)         switch (i) {
(7)             case 0: i++;
(8)                                     break here;
(9)             case 1:
(10)            case 2: i--;
(11)                                     break;
(12)            default:
(13)                                     i *= 10;
(14)        } //switch
(15)    } //main()
(16)} //class SwitchTest
```

Beispiel 7: Switch-Statement [SwitchTest.java](#)

Anmerkungen:

- Der break-Anweisung kann eine Marke nachgestellt werden, die angibt auf welcher Schachtelungsebene fortgesetzt werden soll ([siehe Java Language Specification](#)). Hierbei sind jedoch nur Rückwärtsreferenzen zugelassen.
- Soll ein case-Körper für mehrere Alternativen gelten, so müssen diese explizit mit einleitendem case aufgeführt werden. Pro case ist jeweils nur genau eine konstante Bedingung zugelassen.
- Obwohl switch-Statement syntaktisch unverändert von C/C++ übernommen wurde, existieren in Java zusätzliche Restriktionen, welche die Kompatibilität einschränken.
So muß der Rumpf einer switch-Anweisung die zugehörigen case-Alternativen direkt enthalten; Konstruktionen wie *duff's devices* [siehe Java Language Specification](#) sind daher nicht compilierbar.
- Die switch-Bedingung muß vom Typ byte, short, char oder int sein.

▲ 2.3.2 Iteration -- for-, do-while-Schleifen

Java bietet die bereits in C/C++ eingeführten Schleifenkonstrukte

- for ([siehe Java Language Specification](#))
- while ([siehe Java Language Specification](#))
- do ([siehe Java Language Specification](#))
- break und continue ([siehe Java Language Specification](#))

an.

Die **for-Struktur** besteht aus drei optionalen Komponenten: Initialisierung(en), Fortsetzungsbedingung(en) und Wertaktualisierung(en) (auch Reinitialisierung(en)).

Ebenso wie in C/C++ sind diese durch Semikola voneinander abgetrennt.

Im Initialisierungs- und Fortsetzungsteil sind mehrere, durch Komma separierte, Ausdrücke zugelassen. Im Fortsetzungsbedingungsteil hingegen nicht! Hier müssen mehrere Bedingungen durch logisches *Und* (&&) verbunden werden.

```

(1)public class ForTest {
(2)    public static void main(String[] args) {
(3)        for (int i=1, j=-2; i <= 10 && j <=0 ; i+=2, j++)
(4)            System.out.println("i = "+i+" j = "+j);
(5)    } //main()
(6)} //class ForTest

```

Beispiel 8: Eine for-Schleife [ForTest.java](#)

Alle Schleifen können vorzeitig, d.h. trotz gültiger Fortsetzungsbedingung(en) wahlfrei mit der break-Anweisung verlassen werden.

Generell versucht die break-Anweisung hinter dem nächstliegenden (d.h. innsteren erreichbaren) schließenden Schleifenkonstrukt fortzusetzen. Der Einsatz dieser Anweisung außerhalb der beschriebenen Strukturen wird per Übersetzungsfehler verhindert.

Zusätzlich können Sprungziele durch Marken explizit benannt werden. Diese *labels* werden durch einen eindeutigen Namen abgeschlossen von einem Doppelpunkt symbolisiert.

Es sind jedoch nur „Rückwärtssprünge“ möglich. ([siehe Java Language Specification](#))

```

(1)public class BreakTest {
(2)    public static void main(String[] args) {
(3)        myLabel:
(4)        for (int i=1; i<=100; i++) {
(5)            for(int j=1; j<=100; j++) {
(6)                if (i*j >= 42) {
(7)                    System.out.println("exiting loop");
(8)                    break myLabel;
(9)                } //if
(10)            } //for
(11)        } //for
(12)        System.out.println("continuing...");
(13)    } //main()
(14)} //class BreakTest

```

Beispiel 9: Verlassen einer Schleife mit break [BreakTest.java](#)

Im Gegensatz zu vielen prozeduralen Programmiersprachen verfügt Java über kein goto-Statement welche beliebige Sprünge erlauben würde.

Allerdings scheint beim Sprachdesign durchaus an diese Möglichkeit gedacht worden zu sein. Findet sich doch goto in der Aufzählung der reservierten Schlüsselworte ([siehe Java Language Specification](#)) und im Index ([siehe Java Language Specification](#)).

Die Referenzimplementierung des Javacompilers von SUN nutzt das Schlüsselwort lediglich um den illegalen Beginn eines Ausdrucks zum Übersetzungszeitpunkt anzuzeigen.

Zum vorzeitigen Rücksprung aus dem Schleifenkörper ist das das continue-Statement definiert.

Nach seiner Ausführung erfolgt unverzüglich die Auswertung der Fortsetzungsbedingung, unter Auslassung aller folgenden Anweisungen des aktuellen Blocks.

Analog der break-Anweisung kann die Schachtelungstiefe in der fortgefahren werden soll durch Angabe eines Labels gesteuert werden. ([siehe Java Language Specification](#))

Kopf- und Fußgesteuerte Schleifen, werden mit der while ... do-Konstruktion analog zu C/C++ realisiert.

Auch hier wird der Typ der Bedingung bereits zur Übersetzungszeit auf boolean geprüft.

Häufige Anwendungsform von Schleifen ist die Traversierung einer Objektmenge.

Das nachfolgende Beispiel zeigt die „klassische“ Vorgehensweise zur Ausgabe aller Elemente einer Aufzählung.

Hierzu wird zunächst Elementanzahl ermittelt und anschließend in einer for-Schleife, beginnend mit der kleinsten Indexnummer (0) bis zur ermittelten Obergrenze inkrementiert. An jeder Indexposition wird das unter dieser Ordnungsnummer abgelegte Element ausgegeben.

Der Vorgehensweise unterliegt den beiden Grundannahmen, daß die Werte einerseits kontinuierlich (d.h. keine Indexposition ist unbesetzt) abgelegt sind. Und zweitens, daß die Wertemenge ab der Indexposition 0 aufsteigend abgelegt ist. Beide Annahmen können für die durch die Java-Standardklasse [Vector](#) realisierte Aufzählung als erfüllt angenommen werden.

```

(1)import java.util.Vector;
(2)
(3)public class NaiveIteration {
(4)    public static void main(String[] args) {
(5)        Vector v = new Vector();
(6)        v.add(new String("Berta"));
(7)        v.add(new String("Anna"));
(8)        v.add(new String("Cäsar"));
(9)
(10)        for (int i=0; i<v.size(); i++) {
(11)            System.out.println(v.get(i));
(12)        } //for
(13)    } //main()
(14)} //class NaiveIteration

```

Beispiel 10: Naive Traversierung einer Objektmenge [NaiveIteration.java](#)

Die Lösung leistet zwar das gewünschte, jedoch wirkt die Schleifenkonstruktion unnötig komplex. Überdies zwingt sie den Programmierer Daten abzuspeichern (in Form des Schleifenzählers `i` sowie der implizit angeforderten unbenannten Speicherstelle zur Ablage der Elementanzahl), die nur zum Zweck der Mengentraversierung benötigt werden.

Um die Umsetzung dieser häufig auftretenden Standardsituation zu vereinfachen bietet die Standard-API die Schnittstelle [Iterator](#) an. Sie entbindet den Programmierer von der expliziten Ermittlung der Elementanzahl, sowie der indexgebundenen Traversierung.

Das nachfolgende Beispiel zeigt die Integration des Iterator-basierten Ansatzes für die bekannte Mengentraversierungsaufgabe

```
(1)import java.util.Iterator;
(2)import java.util.Vector;
(3)
(4)public class IteratorTest {
(5)    public static void main(String[] args) {
(6)        Vector v = new Vector();
(7)        v.add(new String("Berta"));
(8)        v.add(new String("Anna"));
(9)        v.add(new String("Cäsar"));
(10)
(11)        for (Iterator i = v.iterator() ; i.hasNext() ;) {
(12)            System.out.println(i.next());
(13)        } //for
(14)    } //main()
(15)} //class IteratorTest
```

Beispiel 11: Iterator-basierte Traversierung einer Objektmenge [IteratorTest.java](#)

Die Lösung enthebt den Programmierer zwar vom Aufwand die Elementanzahl explizit zu ermitteln und einem Speicherplatz zuzuweisen, jedoch wird mit dem Iterator-Objekt immernoch benannter Speicher (in Form der Variable `e`) angefordert, der ausschließlich der schleifeninternen Logik dient.

Erweiterte Schleifen-Syntax

Zur Behebung des Übelstandes der unnötigen expliziten Informationsermittlung im vorigen Beispiel existiert seit Java Version 1.5 eine abkürzenden Schreibweise für Mengentraversierungen.

```
(1)import java.util.Enumeration;
(2)import java.util.Vector;
(3)
(4)public class NewIteration {
(5)    public static void main(String[] args) {
(6)        Vector v = new Vector();
(7)        v.add(new String("Berta"));
(8)        v.add(new String("Anna"));
(9)        v.add(new String("Cäsar"));
(10)
(11)        for (Object o : v) {
(12)            System.out.println( o );
(13)        } //for
(14)    } //main()
(15)} //class NewIteration
```

Beispiel 12: Traversierung einer Objektmenge [NewIteration.java](#)

Das Beispiel zeigt die alternative Schreibweise, welche keine unnötigen, d.h. im Schleifenrumpf nicht benötigten, Daten ermittelt. Der Ausdruck innerhalb der `for`-Klammern typisiert die Elemente der Menge `v` als `Object` und weist sie temporär (konkret: für jeden Schleifendurchlauf einen Wert) der Variable `o` zu.

Die Entnahme aus der Objektmenge erfolgt unter Nutzung des allgemeinen Typs `Object` anstatt direkt auf den konkreten Typ `String` zurückzugreifen. Diese Asymmetrie erklärt sich aus Nutzung der Klasse `Vector` ohne Verwendung der in der Programmiersprache vorhandenen Generizitätsmechanismen. Nähere Ausführungen zu den damit verbundenen Möglichkeiten finden sich im Kapitel [Parametrische Polymorphie/Generics](#).

▲ **2.3.3 Ausnahmen und ihre Behandlung -- Exception Handling**

Die Ausnahmebehandlung von Java ist konzeptionell und syntaktisch eng an das Pendant des ANSI-C++-Standards angelehnt. ([siehe Java Language Specification](#)).

Drei generelle Vorteile ergeben sich durch Verwendung von Ausnahmen:

- Separierung von Code und Fehlerbehandlung ([siehe Java Tutorial](#))
- Propagierung des Ausnahmezustandes; d.h. Behandlung muß nicht lokal in der fehlerauslösenden Routine erfolgen ([siehe Java Tutorial](#))
- Bildung von Fehlerklassen durch Gruppierung gleichartiger Ausnahmen ([siehe Java Tutorial](#))

Die generelle Syntax kann wie folgt beschrieben werden:

```
try {
    //statements which may throw an exception
    //... or create and throw an exception object manually
} catch (exceptionFoo e) {
    //handling of exception of type exceptionFoo
    //exception object is named e
} catch (exceptionBar e) {
    //handling of exception of type exceptionBar
    //exception object is named e
} catch (Exception e) {
    //handling all exceptions not handled by specialized handlers yet
    //exception object is named e
} finally {
    //executed regardless the execution state of the previous try block
} //finally
```

```
(1)public class ExceptionHandlingTest1 {
(2)    public static void main(String[] args) {
(3)        for (int i=2; i>=0; i--)
(4)            System.out.println(42/i);
(5)    } //main()
(6)} //class ExceptionHandlingTest1
```

Beispiel 13: Exception auslösender Code [ExceptionHandlingTest1.java](#)

Das Beispiel illustriert das Auftreten einer *java.lang.ArithmeticException*, verursacht durch die Division durch Null.

Alle in einen try-Block eingeschlossenen Anweisungen werden „überwacht“ ausgeführt. Das bedeutet, es erfolgt kein Programmabbruch beim Auftreten einer Ausnahme, sondern der direkte Anspruch eines *exception handlers*. Exception Handler werden durch catch-Blöcke implementiert. Der Typ der zu behandelnden Exception wird als Übergabeparameter angegeben. Das Laufzeitsystem sorgt für Auswahl der korrekten (d.h. zuständigen) Behandlungsroutine.

Alle Ausnahmen erben von der Klasse [Exception](#). Hierdurch kann auch ein Vorgabe-Exception-Handler installiert werden. Das folgende Beispiel zeigt diesen im Anschluß an die Behandlungsroutine der *ArithmeticException* Ausnahme. Eine Übersicht der in der Java-API definierten Ausnahme findet sich in: ([siehe Java API Specification](#)) Der Ausdruck `catch (Exception e)` übernimmt hierbei die Rolle des aus C++ bekannten `catch(...)`.

```
(1)public class ExceptionHandlingTest2 {
(2)    public static void main(String[] args) {
(3)        try {
(4)            for (int i=2; i>=0; i--)
(5)                System.out.println(42/i);
(6)        } catch (java.lang.ArithmeticException e) {
(7)            System.out.println("an arithmetic exception was thrown -- aborting
program");
(8)            System.out.println("Exception's message "+e.getMessage() );
(9)            System.out.println("Stack trace: ");
(10)           e.printStackTrace();
(11)        }
(12)        catch (Exception e) {
(13)            System.out.println("an exception was thrown -- aborting program");
(14)        } finally {
(15)            System.out.println("finished try block");
(16)        } //finally
(17)    } //main()
(18)} //ExceptionHandlingTest2
```

Beispiel 14: Ausnahmebehandlung [ExceptionHandlingTest2.java](#)

Das Programm liefert die Ausgabe:

```
$java ExceptionHandlingTest1
21
42
```

```

        an arithmetic exception was thrown
    finished try block

```

Der optional angebbare `finally`-Block wird immer ausgeführt, unabhängig davon ob der von `try` umschlossene Anweisungsteil erfolgreich (d.h. fehlerfrei) oder durch Exception (oder auch sonstige Sprungmechanismen wie `break`) verlassen wurde. Er bietet die Gelegenheit nach erfolgter Ausnahmebehandlung „Aufräumarbeiten“, wie das Schließen möglicherweise noch geöffneter Dateien, durchzuführen.

Nützliche Zusatzinformationen über die Art des aufgetretenen Ausnahmeereignisses können durch die Methoden `getMessage()` und `printStackTrace()` abgefragt werden. Beide Methoden sind von `java.lang.Throwable`, der Superklasse von `java.lang.Exception`, ererbt und stehen daher auf allen Ausnahmeobjekten zur Verfügung.

Ausnahmen die als Subklassen von `RuntimeException` realisiert sind müssen nicht explizit deklariert oder aufgefangen werden, da sie von *Instruktionen der virtuellen Maschine* erzeugt werden. Dies stellt einen Widerspruch zur erhobenen Forderung dar, daß alle Ausnahmen, die Subklassen von `Throwable` sind, explizit zu deklarieren und aufzufangen sind!

Eine korrekte Deklaration wäre jedoch unter praktischen Gesichtspunkten nicht praktikabel, da beispielsweise ein Fehler des Typs `InternalError` potentiell durch jede Methode erzeugt werden kann.

Abgesehen davon verhalten sich jedoch Laufzeit-Ausnahmen jedoch wie „normale“ Exceptions. Begründet durch Abwesenheit einer expliziten Deklaration sieht man auch -- außer in raren und begründeten Ausnahmefällen -- vom Auffangen und Behandeln durch den Programmierer ab.

Erzeugen eigener Ausnahmeereignisse

Neben durch das Laufzeitsystem generierten Ausnahmeereignissen können auch innerhalb des Programmcodes gezielt Exceptions durch den Anwender generiert werden. Hierfür existiert das `throw`-Statement.

```

(1)public class OwnException1 {
(2)    public static void main(String[] args) {
(3)        try {
(4)            System.out.println("throwing an arithmetic exception...");
(5)            throw new ArithmeticException();
(6)            //never gets here
(7)        } //try
(8)        catch (Exception e) {
(9)            System.out.println(e.toString() + " exception caught");
(10)        } //catch
(11)    } //main()
(12)} //class OwnException1

```

Beispiel 15: Anwenderdefiniert ausgelöste Ausnahme [OwnException1.java](#)

Das Codebeispiel zeigt das manuelle Erzeugen einer `ArithmeticException`.

Anmerkung: Nach `throw` angegebene Anweisungen können niemals erreicht werden, und führen bereits zum Übersetzungszeitpunkt zu einer entsprechenden Fehlermeldung.

Das bestehende vorgegebene System der Exceptions kann durch den Programmierer jederzeit um eigendefinierte Ausnahmen erweitert werden. Voraussetzung hierfür ist die Definition einer neuen Ausnahmeklasse; dies geschieht (im einfachsten Falle) durch Erben von `java.lang.Exception`.

Das nachfolgende Beispiel illustriert dies:

```

(1)public class OwnException2 {
(2)    public static void main(String[] args) {
(3)        try {
(4)            System.out.println("throwing myException...");
(5)            throw new myException();
(6)        } catch (Exception e) {
(7)            System.out.println(e.toString() + " exception caught");
(8)        } //catch
(9)    } //main()
(10)} //class OwnException2
(11)
(12)class myException extends Exception {
(13)} //class myException

```

Beispiel 16: Eigendefinierte Ausnahme [OwnException2.java](#)

Für alle Ausnahmen die nicht lokal behandelt werden, kann durch Angabe der `throws`-Liste die Ausnahmenbehandlung an den Aufrufer weitergereicht werden.

```

(1)public class OwnException3 {
(2)    public static void main(String[] args) {
(3)        try {
(4)            exceptionProne();
(5)        } catch (myException myE) {
(6)            System.out.println("Exception of type myException caught!");
(7)        } //catch
(8)    } //main()
(9)
(10)    public static void exceptionProne() throws myException {
(11)        throw new myException();
(12)    } //exceptionProne()
(13)} //class OwnException3
(14)
(15)class myException extends Exception {
(16)} //class myException

```

Beispiel 17: Propagierung der Ausnahmebehandlung [OwnException3.java](#)

Im Beispiel wird die eigendefinierte Ausnahme myException nicht innerhalb der Methode exceptionProne behandelt, sondern an den Aufrufer -- in diesem Beispiel die main-Methode -- zur Behandlung weitergereicht. Der Übersetzer prüft hier das Vorhandensein eines try-catch-Blockes innerhalb von main ab. Das folgende Beispiel zeigt eine Erweiterung des vorhergehenden, dergestalt, daß auch die main-Methode mit einem throws versehen ist. In diesem Fall wird das Ausnahmeereignis an das Laufzeitsystem weitergereicht; welches in der Konsequenz die Ausführung terminiert.

Generell gilt in Java die *catch or throw*-Regel, die besagt, daß ein Ausnahmeereignis entweder aufgefangen und behandelt (*catch*) oder weitergereicht (*throw*) werden muß.

```

(1)public class OwnException4 {
(2)    public static void main(String[] args) throws myException {
(3)        throw new myException();
(4)    } //main()
(5)} //class OwnException4
(6)
(7)class myException extends Exception {
(8)} //class myException

```

Beispiel 18: Propagierung eines Ausnahmeereignisses an das Laufzeitsystem [OwnException4.java](#)

Auch die Kombination der beiden vorgestellten Ansätze ist möglich ...

Im abschließenden Codebeispiel wird neben der lokalen Ausnahmebehandlung (catch-Block innerhalb exceptionProne()) auch eine Behandlung innerhalb der aufrufenden Methode (main) durchgeführt. Hierzu wird das aufgefangene Ausnahmeereignis innerhalb der Behandlungsroutine erneut mittels throw ausgelöst.

```

(1)public class OwnException5 {
(2)    public static void main(String[] args) {
(3)        try {
(4)            exceptionProne();
(5)        } catch (myException e) {
(6)            System.out.println("exception myException caught within main method");
(7)        } //catch
(8)    } //main()
(9)
(10)    public static void exceptionProne() throws myException {
(11)        try {
(12)            throw new myException();
(13)        } catch (myException e) {
(14)            System.out.println("exception myException caught within method
exceptionProne");
(15)            System.out.println("re-throwing...");
(16)            throw e;
(17)        } //catch
(18)    } //exceptionProne()
(19)} //class OwnException5
(20)
(21)class myException extends Exception {
(22)} //class myException

```

Beispiel 19: Behandlung und Weiterreichung einer Ausnahme [OwnException5.java](#)

Zwangsbedingungen

Zur Steigerung der Qualität des entstehenden Systems gestattet Java die Definition von *Zwangsbedingungen* (*Assertion*), deren Einhaltung während der Ausführung geprüft werden kann. Ist eine solche Bedingung nicht erfüllt, so erfolgt ein Programmabbruch.

Im Unterschied zur Möglichkeit durch Selektionsausdrücke die Rückgabewerte von Methodenaufrufen auszuwerten oder durch Ausnahmebehandlung gezielt und benutzerdefiniert auf Fehlersituationen zu reagieren bieten die Zwangsbedingungen sowohl eine gegenüber den beiden genannten Ansätzen signifikant kompaktifizierte Syntax an, die gleichzeitig weniger Freiheitsgrade in der Behandlung der Fehlersituation bietet.

Zusätzlich kann die Prüfung von Zwangsbedingungen zur Laufzeit statisch durch einen Schalter der Ausführungsumgebung gesteuert werden. Auf dieser Basis eignen sie sich gut für die Formulierung verschiedenster Konsistenzprüfungen, die später im Produktivbetrieb zur Steigerung der Ausführungsgeschwindigkeit deaktiviert werden können.

Die allgemeine Syntax einer Zwangsbedingung lautet:

```
assert Boole'scher-Ausdruck (: Ausdruck)opt
```

Generell werden Zwangsbedingungen durch das Schlüsselwort `assert` eingeleitet. Darauf folgt ein Boole'scher Ausdruck, der erfüllt sein muß. Liefert die Auswertung dieses Ausdrucks den Wahrheitswert `false`, so erfolgt der Programmabbruch. Ist zusätzlich, nach dem separierenden Doppelpunkt, ein Ausdruck angegeben, so wird dieser zur Konstruktion eines `AssertionError`-Objekts herangezogen.

Aus den zugelassenen Parametertypen eines solchen Objekts ergeben sich die möglichen angebbaren Ausdruckstypen als: `boolean`, `char`, `double`, `float`, `int`, `long` und `Object`.

Das nachfolgende Beispiel zeigt den Einsatz einer einfachen Zwangsbedingung, deren Fehlschlag ohne anwenderdefinierte Konstruktion `AssertionError`-Objekts behandelt wird:

```
(1)public class AssTest1 {
(2)    public static void main(String[] args) {
(3)        Object o = null;
(4)        //...
(5)        assert o != null;
(6)    } //main()
(7)} //class AssTest1
```

Beispiel 20: Einfache Zwangsbedingung [AssTest1.java](#)

Das Beispiel deklariert eine Variable `o` als Ausprägung der Klasse `Object` und initialisiert diese mit `null`. Im späteren Verlauf der Ausführung soll geprüft werden, ob zwischenzeitlich eine Initialisierung erfolgt ist. Ist dies nicht der Fall, so ist eine Programmfortsetzung nicht sinnvoll. Diese Prüfung, verbunden mit der impliziten Forderung den Ablauf bei ihrer Nichterfüllung zu terminieren, ist als Zwangsbedingung realisiert.

Zur Verwendung der Zwangsbedingungen ist die Übersetzung mindestens konform zur Sprachversion 1.4 notwendig. Aktiviert wird diese Quellcodeinterpretation durch Übergabe des Wertes „1.4“, oder höher, als Parameter des Übersetzerschalters `-source`. Wird ein Wert kleiner als 1.4 gewählt, so wird das Schlüsselwort `assert` nicht als solches interpretiert, sondern kann als Identifier zur Benennung von Klassen, Attributen oder Methoden benutzt werden.

Im selben Sinne ist es ebenso zwingend erforderlich die Prüfung von Zwangsbedingung innerhalb der Laufzeitumgebung durch den Schalter `-enableassertions` zu aktivieren. Andernfalls werden alle `assert`-Anweisungen ungeprüft ignoriert. Aus dieser Forderung ergibt sich, daß mindestens Version 1.4 der Laufzeitumgebung benötigt wird um die Interpretation von Zwangsbedingungen zu aktivieren.

Die Ausführung des Beispiels liefert daher, bei aktiviertem Schalter die Ausgabe:

```
Exception in thread "main" java.lang.AssertionError
    at AssTest1.main(AssTest1.java:5)
```

Zur Einschränkung der Zwangsbedingungsauswertung gestattet die Laufzeitumgebung die Spezifikation derjenigen Klassen, für die diese Bedingungen ausgewertet werden sollen. Hierzu werden nach dem Schlüsselwort `enableassertions`, abgetrennt durch einen Doppelpunkt, die Namen der zu berücksichtigenden Klassen angegeben werden. Für das obenstehende Beispiel sind daher die Aufrufe `java -enableassertions AssTest1` und `java -enableassertions:AssTest1 AssTest1` äquivalent.

Ebenfalls durch Kommandozeilenschalter kann die (De-)Aktivierung der Prüfung der in den Standard-API-Klassen formulierten Zwangsbedingungen gesteuert werden. Hierfür stehen die Schalter `-enablesystemassertions` bzw. `-disablesystemassertions`. Die Möglichkeiten der klassengenauen Steuerung stehen jedoch in diesem Falle nicht zur Verfügung.

Die Möglichkeit der Übergabe von Parametern an das automatisiert durch das Laufzeitsystem erzeugte `AssertionError`-Objekt bietet sich insbesondere zur Dokumentation der Abbruchursache an. So zeigt das nachfolgende Beispiel prüft durch Aufruf der Methode `exists` der Klasse `File` ob die Datei `test` im aktuellen Dateisystemkatalog angelegt ist. Ist dies nicht erfüllt, so wird eine festgelegte Zeichenkette dem Konstruktor von `AssertionError` übergeben. Die Zeichenkette kann, als Ausprägung der Klasse `String` zur Konstruktion eines Fehlerobjektes verwendet werden, da sie gemäß der Typrestriktion eine gültige Instanz von `Object` repräsentiert. Gleichzeitig kann die Methode `exists` als Ausdruck nach dem `assert`-Schlüsselwort angegeben werden, da die Ausführung der Methode einen Rückgabewert vom Typ `boolean` liefert und damit der Gesamtausdruck als

Wahrheitswert ausgewertet werden kann.

```
(1)import java.io.File;
(2)
(3)public class AssTest2 {
(4)    public static void main(String[] args) {
(5)        //...
(6)        assert (new File("test")).exists() : "File 'test' does not exist";
(7)    } //main()
(8)} //class AssTest2
```

Beispiel 21: Zwangsbedingung mit anwenderdefinierter Fehlerobjekt [AssTest2.java](#)

Zwar sollte nach Nichterfüllung einer Zwangsbedingung die Applikationsausführung beendet werden, wie es auch im Standardfalle durch das Laufzeitsystem geschieht. Jedoch kann dieses Verhalten mit Mitteln des Ausnahmebehandlung unterbunden werden. Hierzu muß eine Zwangsbedingungsauwertung in einen try-Block einbettet werden. Findet sich im zugeordneten catch-Bereich eine Klausel für den AssertionError so wird diese ausgeführt und anschließend die Programmverarbeitung normal fortgesetzt. Das nachfolgende Beispiel zeigt diese Anwendung.

```
(1)import java.io.File;
(2)
(3)public class AssTest3 {
(4)    public static void main(String[] args) {
(5)        //...
(6)        try {
(7)            assert false : "this assertion fails definitely";
(8)        } catch (AssertionError ae) {
(9)            System.out.println("caught assertion error");
(10)        } //catch
(11)        System.out.println("continuing after error ");
(12)    } //main()
(13)} //class AssTest3
```

Beispiel 22: Abfangen einer fehlschlagenden Zwangsbedingung [AssTest3.java](#)

Das im Beispiel gezeigte Verhalten ist zwar syntaktisch korrekt und wird auch im angestrebten Sinne ausgeführt, sollte jedoch nur mit Bedacht angewandt werden, da es die Semantik einer Zwangsbedingung aufweicht. Insgesamt empfiehlt sich die Verwendung von Zwangsbedingungen lediglich für eine engumrissene Klasse von Fehlern, wie Nachbedingungen von Methoden, die einen internen Systemzustand dokumentieren, der durch keine Modifikation in einen konsistenten Zustand überführt werden könnte, er eine Ausführungsfortsetzung sinnvoll erscheinen läßt. Insbesondere solche, die durch Anwenderinteraktion begründet sind (wie fehlerhafte Parameterübergabe, etc.) sollten durch Ausnahmen behandelt werden.

▲ 2.4 Von komplexen zu objektorientierten Datenstrukturen

▲ 2.4.1 Arrays

Wie fast alle prozeduralen Programmiersprachen unterstützt auch Java die Zusammenfassung beliebiger gleichartiger Elemente zu geordneten, nicht zwingend duplikatfreien, Mengen; die sog. *Feldern*, *Arrays* oder *Vektoren*.

Arrays in Java sind nicht Bestandteil des [built-in Typsystems](#), sondern bereits als [Klasse innerhalb der Java API](#) realisiert.

Wie in anderen Programmiersprachen üblich stellen Javaarrays einen zusammenhängen kontinuierlichen Speicherbereich dar.

Bereits innerhalb der Standardsignatur der main-Methode wird ein Array verwendet:
`public static void main(String[] args).`

Einige Charakteristika:

- Statische Größenfestlegung.
 Arraygrenzen können nach Definition nicht mehr verändert werden.
 Vorgriff: Dies ist mit der API Klasse [Vector](#) möglich.
- Arrays sind (*first class*) Objekte.
 => Sie müssen durch den Programmierer explizit mit new erzeugt werden.
- Die Inhaltselemente sind immer beginnend mit 0 nummeriert.
- Zugriffe über die Arraygrenzen hinaus werden durch die `ArrayIndexOutOfBoundsException` abgefangen.
- Definition anonymer (d.h. namenloser) Arrays möglich.

- Definition „mehrdimensionaler“ Arrays möglich.
...dazu später mehr.
- Array können bereits während der Definition initialisiert werden.
Hierdurch können auch dynamisch allokierte Arrays realisiert werden.
- Zeichenketten (Strings) sind *nicht* als *Array of Character*, sondern als [eigene API-Klasse](#), realisiert.
- Der Zugriff auf einzelne Arrayelemente erfolgt über Index (=berechenbare Adresse).
`a[i]` greift auf das `i+1`-te Element des Arrays `a` zu.
- Arraykomponenten werden bei der Erzeugung mit `new` initialisiert.
Im Falle primitiver Datentypen wird als Vorgabewert `0` für alle numerischen Datentypen, `false` für boolean und Unicode `\u0000` für char gesetzt. Objektwertige Typen werden durch ihren Konstruktor initialisiert.

Syntax:

```

ArrayCreationExpression:
    new PrimitiveType DimExprs DimsOptional
    new TypeName DimExprs DimsOptional
    new PrimitiveType DimsOptional ArrayInitializer
    new TypeName Dims ArrayInitializer

```

```

DimExprs:
    DimExpr
    DimExprs DimExpr

```

```

DimExpr:
    [ Expression ]

```

```

Dims:
    [ ]
    Dims [ ]

```

```

(1)public class Array1 {
(2)    public static void main(String[] args) {
(3)        int                firstArray[]    = new int[10];
(4)        int[]              secondArray     = new int[10];
(5)        double[]           thirdArray      = {3.14, 2+5, 42.0};
(6)
(7)        System.out.println("length of firstArray="+ firstArray.length );
(8)        System.out.println("length of secondArray="+ secondArray.length );
(9)        System.out.println("length of thirdArray="+ thirdArray.length );
(10)       for (int i=0; i<thirdArray.length; i++)
(11)           System.out.println("thirdArray["+i+"]="+thirdArray[i]);
(12)
(13)       System.out.println("length of anonymous array="+ (new byte[] {1,2,3}).length);
(14)    } //main()
(15)} //class Array1

```

Beispiel 23: Arrayerzeugung und Initialisierung [Array1.java](#)

Der Code aus Beispiel 16 definiert zunächst zwei `int`-Array der Größe 10. Die beiden Syntaxvarianten sind (wie aus C/C++ bekannt) äquivalent.

`thirdArray` wird bereits während der Definition mit `double`-Werten initialisiert. Die Größe muß nicht explizit angegeben werden, die errechnet sich automatisch aus der Anzahl der angegebenen Ausdrücke.

Die letzte Definition eines Arrays im Beispiel erfolgt anonym. Der so erzeugte Array steht nach Verlassen des `System.out.println`-Ausdrucks nicht mehr zur Verfügung.

Arrays deren Elemente eigendefinierte Typen sind werden entsprechend mit *eigenerTyp[] arrayName ...* definiert.

Mehrdimensionale Arrays ...

werden nicht explizit unterstützt, sondern als Arrays von Arrays behandelt.

```

(1)public class Array2 {
(2)    public static void main(String[] args) {
(3)        int[][] multiDimensional = new int[2][3];
(4)
(5)        int dimR = multiDimensional.length;
(6)        int dimC = multiDimensional[0].length;
(7)
(8)        System.out.println("Length of multiDimensional="+ dimR);
(9)        System.out.println("Length of multiDimensional[0]="+dimC );
(10)
(11)       for(int i=0; i<dimR; i++)
(12)           for(int j=0; j<dimC; j++)
(13)               multiDimensional[i][j] = i*dimC + j+1;
(14)
(15)       System.out.println("Content of multiDimensional:");
(16)       for(int i=0; i<dimR; i++) {
(17)           for(int j=0; j<dimC; j++) {
(18)               System.out.print(multiDimensional[i][j]+ " ");
(19)           } //for

```

```

(20)                System.out.println();
(21)                } //for
(22)    } //main()
(23)} //class Array2

```

Beispiel 24: Mehrdimensionale Arrays [Array2.java](#)

BildschirmAusgabe:

```

$java Array2
Lenght of multiDimensional=2
Lenght of multiDimensional[0]=3
Content of multiDimensionl:
1 2 3
4 5 6

```

Die Angabe mehrer Dimensionsgrößen bei der Definition stellt eine Kurzform für die verschachtelte Variante dar. So ist das nachfolgende Codefragment äquivalent zum vorhergehenden Beispiel:

```

(1)public class Array3 {
(2)    public static void main(String[] args) {
(3)        int[][] multiDimensional = new int[2][];
(4)        for (int i=0; i<multiDimensional.length; i++)
(5)            multiDimensional[i] = new int[3];
(6)
(7)        int dimR = multiDimensional.length;
(8)        int dimC = multiDimensional[0].length;
(9)
(10)       System.out.println("Lenght of multiDimensional="+ dimR);
(11)       System.out.println("Lenght of multiDimensional[0]="+dimC );
(12)
(13)       for(int i=0; i<dimR; i++)
(14)           for(int j=0; j<dimC; j++)
(15)               multiDimensional[i][j] = i*dimC + j+1;
(16)
(17)       System.out.println("Content of multiDimensionl:");
(18)       for(int i=0; i<dimR; i++) {
(19)           for(int j=0; j<dimC; j++) {
(20)               System.out.print(multiDimensional[i][j]+ " ");
(21)           } //for
(22)       System.out.println();
(23)       } //for
(24)   } //main()
(25)} //class Array3

```

Beispiel 25: verschachtelte mehrdimensionale Arraydefinition [Array3.java](#)

Die Definition

```
int[][] multiDimensional = { {1,2,3}, {100,200,300} };
```

Definiert einen Array der zwei Array, jeweils der Länge 3, enthält, und initialisiert diese mit den angegebenen Werten.

Unterscheiden sich die beiden implizit spezifizierten Arrays in der Größe, so definiert die Anzahl des ersten angegebenen Arrays die Elementzahl (Zeilenlänge) für alle folgenden (Zeileneinträge).

Duplizieren von Arrays:

Für die häufig benötigte Anwendung den vollständigen Inhalt eines Arrays in einen zweiten Array gleicher Dimension (en) zu kopieren existiert die clone-Methode.

```

(1)public class Array4 {
(2)    public static void main(String[] args) {
(3)        int[] ia = {1,2,3,4,5};
(4)        int[] ib = new int[ia.length];
(5)
(6)        System.out.println("ia==ib = "+ (ia==ib));
(7)
(8)        System.out.println("content of ia:");
(9)        for(int i=0; i<ia.length; i++)
(10)            System.out.print(ia[i]+" ");
(11)        System.out.println("");
(12)
(13)        System.out.println("content of ib:");
(14)        for(int i=0; i<ib.length; i++)
(15)            System.out.print(ib[i]+" ");
(16)        System.out.println("");
(17)
(18)
(19)        ib = (int[]) ia.clone();
(20)        System.out.println("content of ib after cloning:");
(21)        for(int i=0; i<ib.length; i++)

```

```

(22)             System.out.print(ib[i]+" ");
(23)             System.out.println("");
(24)
(25)             System.out.println("ia==ib = "+ (ia==ib));
(26)
(27)             System.out.println("Setting ia=ib...");
(28)             ia=ib;
(29)             System.out.println("ia==ib = "+ (ia==ib));
(30)         } //main()
(31)} //class Array4

```

Beispiel 26: Duplizierung eines Arrayinhaltes mit der clone-Methode [Array4.java](#)

BildschirmAusgabe:

```

$java Array4
ia==ib = false
content of ia:
1 2 3 4 5
content of ib:
0 0 0 0 0
content of ib after cloning:
1 2 3 4 5
ia==ib = false

```

Das Beispiel definiert zunächst den Array `ia` per expliziter Initialisierung. Ein zweiter Array, `ib` wird mit derselben Länge wie `ia` definiert.

Wie zu erwarten sind die Arrayobjekte verschieden, d.h. sie belegen unterschiedliche Speicherplätze.

Die `clone`-Methode dupliziert den Inhalt des Arrays `ia` und weist in `ib` zu.

In den letzten Zeilen wird der Variable `ia` der Wert von `ib` zugewiesen. Mithin der Array `ia` durch `ib` überschrieben.

Das ursprüngliche `ia` kann damit nicht mehr durch den Programmierer referenziert werden, und wird für den Garbage Collector als freizugeben markiert.

Die [ArrayStoreException](#) wird ausgeworfen, sobald ein Typkonflikt beim Einfügen eines Arrayelementes zur Laufzeit auftritt; statisch erkennbare Typkonflikte werden bereits durch den Übersetzer gemeldet.

```

(1)public class Array5 {
(2)    public static void main(String[] args) {
(3)        Test2[] ta = new Test2[5];
(4)        Test1[] tb = ta;
(5)
(6)        try {
(7)            tb[0] = new Test1();
(8)        } //try
(9)        catch (ArrayStoreException e) {
(10)            System.out.println(e);
(11)        } //catch
(12)    } //main()
(13)} //class Array5
(14)
(15)class Test1 {
(16)} //class Test1
(17)
(18)class Test2 extends Test1 {
(19)} //class Test2

```

Beispiel 27: Typkonflikt beim Einfügen, der zur Auslösung einer `ArrayStoreException` führt [Array5.java](#)

Obwohl der Typ der Arraykomponenten von `tb` als `Test1` deklariert war, führt die Zuweisung innerhalb des `try`-Blockes zu einer `ArrayStoreException`.

Ursache: Durch die Initialisierung mit Elementen des Typs `Test2`, der eine Spezialisierung von `Test1` bildet, wird auch der erwartete Inhaltstyp von `Test1` verändert (konkret: spezialisiert).

▲ 2.4.2 Klassen

Klassen und Objekte stellen das namensgebende Hauptabstraktionsmittel in der objektorientierten Programmierung dar.

Prinzipiell lassen sich aus Sicht der OO-Programmierung drei Arten von Sprachen unterscheiden:

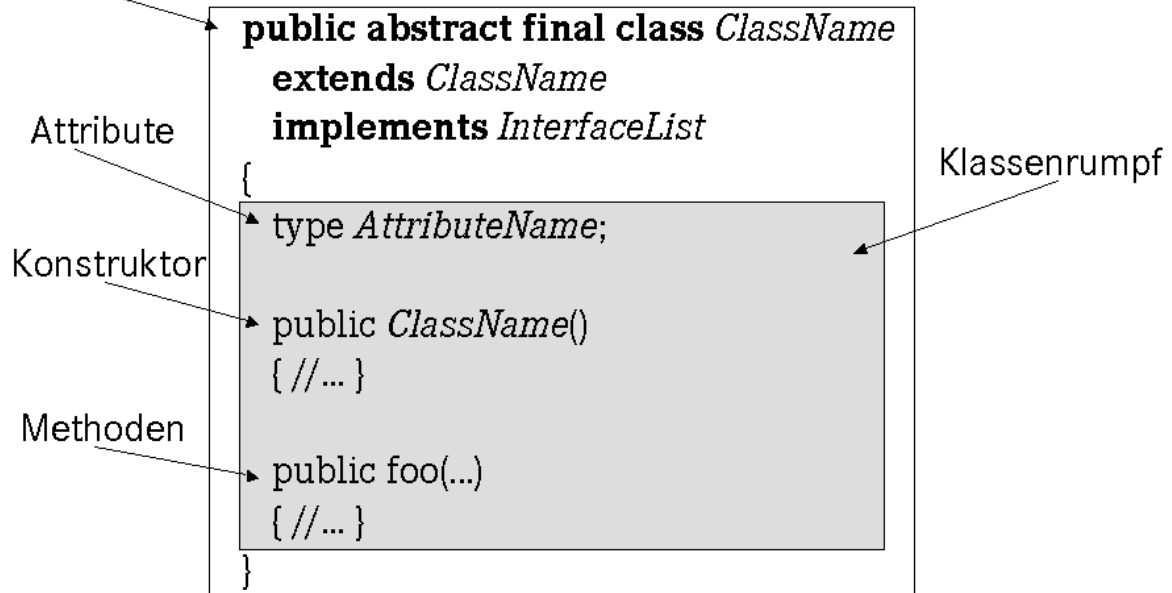
- Klassenlose Sprachen, wie C, Pascal, etc.
- Klassenbasierte hybride Sprachen, die Objekten und Vererbung unterstützen -- jedoch ihre Verwendung nicht auf allen Ebenen durchgängig erzwingen, wie C++ und Java
- (rein) objektorientierte Sprachen, wie SmallTalk, Eiffel, etc.

Gemäß dieser Einordnung ist Java als hybride objektorientierte Sprache anzusehen. Dies rührt hauptsächlich von der Umsetzung des [build-in Typsystems](#) als einfache Werte -- anstatt first class objects -- her. In der Praxis zieht dies jedoch zumeist keine allzugrossen Einschränkungen nach sich.

Die Verwendung von Klassen ist in Java, im scharfen Gegensatz zu C++, zwingend. So ist es nicht möglich ein Programm ohne zumindest eine (Haupt-)Klasse zu schreiben.

Jede Klasse bildet einen abgeschlossenen Sichtbarkeitsbereich (auch: Scope) für die darin definierten [Attribute](#) und [Methoden](#).

Klassendeklaration

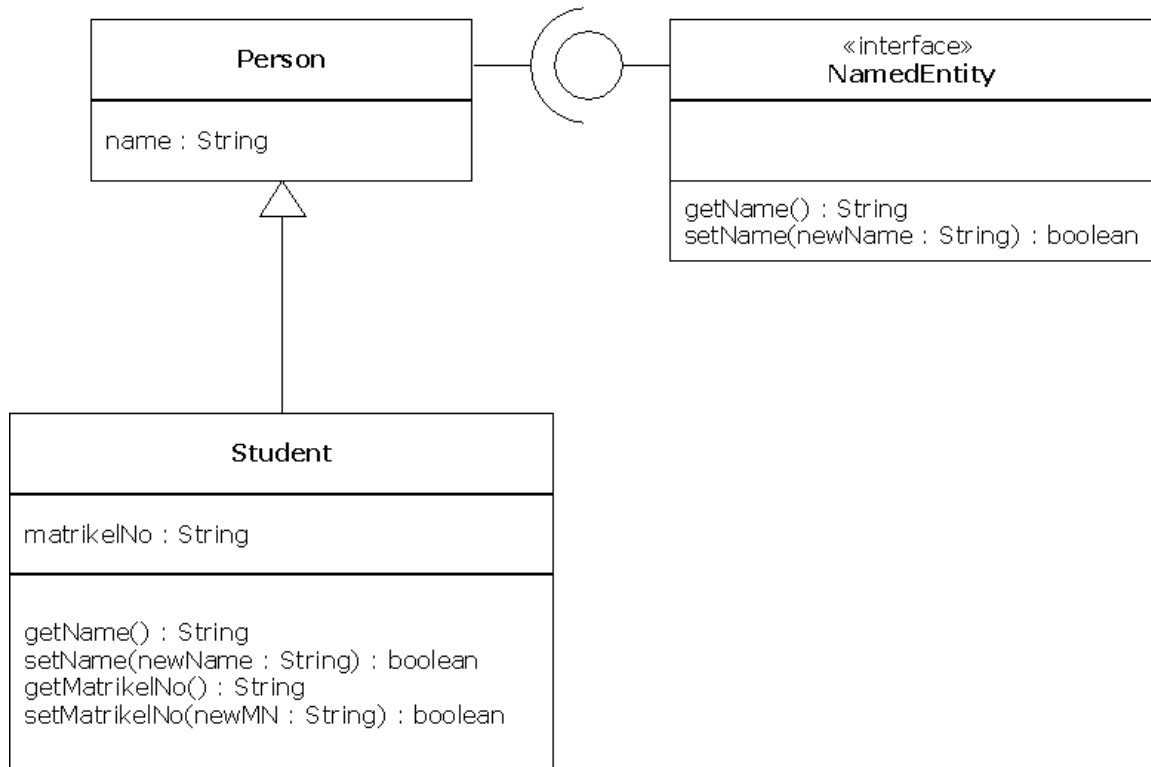


Eine Java-Klasse besteht aus den in der Abbildung dargestellten Teilen:

Klassendeklaration: Sie benennt die Klasse eindeutig, und legt die allgemeinen Charakteristika fest. Das Aussehen der Deklaration kann wie folgt beschrieben werden:

Syntaxelement	Semantik
public	Die Klasse ist allgemein sichtbar. Vorgabegemäß kann eine Klasse nur von Klassen desselben Packages genutzt werden. Durch das Schlüsselwort public wird sie auch außerhalb des umgebenden Packages sichtbar und zugreifbar. Die namensgebende Klasse einer Java-Quellcode-Datei, die auch die main-Methode enthalten kann, muß zwingend als public erklärt sein.
abstract	es können keine Objekte dieser Klasse erzeugt werden. Abstrakte Klassen dienen zur Strukturierung des Entwurfs, um gemeinsame Merkmale verschiedener Klassen zentralisiert ausdrücken zu können.
final	von dieser Klasse kann nicht geerbt werden. Die Entscheidung eine Klasse als final zu deklarieren ist designgetrieben. In der Verwendung der Klasse ergeben sich keine Unterschiede. Jedoch können so Vererbungsäste als abgeschlossen gekennzeichnet werden, um auszudrücken, daß an dieser Stelle keine Weiterentwicklung erfolgen soll. Das Schlüsselwort verhindert einen entsprechenden Versuch durch einen Fehler zum Übersetzungszeitpunkt. Beispiele aus der Java-API: java.io.FileDescriptor , die Wrappertypen: Boolean , Integer , etc.
class <i>ClassName</i>	Name der Klasse
extends <i>ClassName</i>	Die Klasse erbt von der Klasse <i>ClassName</i> In Java ist nur einfache Vererbung zugelassen, daher kann an dieser Stelle auch maximal eine Superklasse spezifiziert werden.
implements <i>InterfaceList</i>	Die Klasse implementiert die in der <i>InterfaceList</i> aufgeführten Schnittstellen. Die Schnittstellennamen werden durch Kommata voneinander abgetrennt.
{ <i>ClassBody</i> }	Ausprogrammierter Klassenrumpf.

Ein umfangreicheres Beispiel:



```

(1)public class Student extends Person implements NamedEntity {
(2)    private String matrikelNo;
(3)
(4)    public Student(String matrikelNo) {
(5)        this.matrikelNo = matrikelNo;
(6)    } //constructor
(7)
(8)    public String getMatrikelNo() {
(9)        return matrikelNo;
(10)    } //getMatrikelNo;
(11)
(12)    public boolean setMatrikelNo(String matrikelNo) {
(13)        if (matrikelNo.compareTo("")==0) {
(14)            this.matrikelNo = matrikelNo;
(15)            return true;
(16)        } else
(17)            return false;
(18)    } //setMatrikelNo()
(19)
(20)    public String getName() {
(21)        return name;
(22)    } //getName()
(23)
(24)    public boolean setName(String newName) {
(25)        if (newName.compareTo("")!=0) {
(26)            name = newName;
(27)            return true;
(28)        } else
(29)            return false;
(30)    } //setName()
(31)
(32)    public String toString() {
(33)        return ("Name: "+this.getName()+"\nMatrikelnummer: "+this.getMatrikelNo() );
(34)    } //toString()
(35)
(36)    public static void main(String[] args) {
(37)        Student mario = new Student("0793022");
(38)        System.out.println( mario.getMatrikelNo() );
(39)        System.out.println("mario.toString() returns:\n"+mario.toString() );
(40)        mario.setName("Mario Jeckle");
(41)        System.out.println("mario.toString() returns:\n"+mario.toString() );
(42)    } //main()
(43)} //class Student
(44)
(45)class Person {
(46)    String name;
(47)} //class Person
(48)
(49)interface NamedEntity {
  
```

```
(50)    String getName();
(51)    boolean setName(String newName);
(52)} //interface NamedEntity
```

Beispiel 28: Beispiel einer ausprogrammierten Klasse [Student.java](#)

Bildschirmausgabe:

```
0793022
mario.toString() returns:
Name: null
Matrikelnummer: 0793022
mario.toString() returns:
Name: Mario Jeckle
Matrikelnummer: 0793022
```

Innere Klassen

Seit Java v1.1 besteht auch die Möglichkeit Klassen innerhalb von Klassen zu definieren.

Merkmale:

- Objekte der inneren Klasse können auf private Daten des erstellenden Objektes zugreifen.
- Innere Klassen unterliegen dem Sichtbarkeitsbereich der umgebenden Klasse, d.h. sie sind vor Zugriffen anderer Klassen geschützt.
- Stark ereignisbezogene Anwendungen wie GUI-Bibliotheken nutzen diese Definitionsvariante sehr stark aus.
- Für inner classes wird eine eigenständige class-Datei mit der Namenskonvention *UmgebendeKlasse\$innereKlasse* erzeugt.

Innere Klassen bilden daher kein originäres Sprachmerkmal, sondern werden durch den Compiler in „normale“ Klassen umgesetzt; für die virtuelle Maschine ist diese Mimik (nahezu) transparent.

(Naive) Erweiterung des vorhergehenden Beispiels: Die Matrikelnummer ist als eigenständige Klasse innerhalb von Student2 realisiert:

```
(1)public class Student2 extends Person implements NamedEntity {
(2)    public Student2(String newMatrikelNo) {
(3)        matrikelNo = new MatrikelNo(newMatrikelNo);
(4)    } //constructor
(5)
(6)    class MatrikelNo {
(7)        private String matrikelNo;
(8)
(9)        public MatrikelNo(String newMatrikelNo) {
(10)            matrikelNo = newMatrikelNo;
(11)            if (this.checkMatrikelNo() != true)
(12)                System.out.println("illegal MatrikelNo");
(13)        } //constructor
(14)        public boolean checkMatrikelNo() {
(15)            return( this.matrikelNo.length() == 7 ? true : false);
(16)        } //end checkMatrikelNo()
(17)
(18)        public String getMatrikelNo() {
(19)            return matrikelNo;
(20)        } //getMatrikelNo()
(21)
(22)        public boolean setMatrikelNo(String newMatrikelNo) {
(23)            matrikelNo = newMatrikelNo;
(24)            if (this.checkMatrikelNo() == true)
(25)                return true;
(26)            else
(27)                return false;
(28)        } //setMatrikelNo()
(29)    } //MatrikelNo;
(30)
(31)
(32)    private MatrikelNo matrikelNo;
(33)
(34)    public String getMatrikelNo() {
(35)        return matrikelNo.getMatrikelNo();
(36)    } //getMatrikelNo;
(37)
(38)    public boolean setMatrikelNo(String matrikelNo) {
(39)        return true;
(40)    } //setMatrikelNo()
(41)
(42)    public String getName() {
(43)        return name;
(44)    } //getName()
(45)
(46)    public boolean setName(String newName) {
```

```

(47)         if (newName.compareTo("")!=0) {
(48)             name = newName;
(49)             return true;
(50)         } else
(51)             return false;
(52)     } //setName()
(53)
(54)     public String toString() {
(55)         return ("Name: "+this.getName()+"\nMatrikelnummer: "+this.getMatrikelNo() );
(56)     } //toString()
(57)
(58)     public static void main(String[] args) {
(59)         Student2 mario = new Student2("0793022");
(60)         System.out.println( mario.getMatrikelNo() );
(61)
(62)         Student2 hans = new Student2("1234567X");
(63)     } //main()
(64)} //class Student2
(65)
(66)class Person {
(67)    String name;
(68)} //class Person
(69)
(70)interface NamedEntity {
(71)    String getName();
(72)    boolean setName(String newName);
(73)} //interface NamedEntity

```

Beispiel 29: Realisierung der Matrikelnummer als innere Klasse [Student2.java](#)

Anonyme Klassen:

Als weitere Besonderheit besteht die Möglichkeit die eingebettete Klasse anonym zu definieren. ([siehe Java Language Specification](#))

Syntaktisch folgt die gesamte Klassendefinition auf eine mit new eingeleitete Objekterzeugung.

Unbenannte Klassen verfügen über keinen Konstruktor, da dieser konsequenterweise auch namenlos sein müsste.

Syntax:

```

new BaseClass (Parameters) {
    //inner class methods and data
};

```

Hinweis: Man beachte das (zwingende) Semikolon am Ende des Ausdrucks!

```

(1)public class Anonymous {
(2)    private TestClass test() {
(3)        return new TestClass() {
(4)            public void hello() {
(5)                System.out.println("overridden test!");
(6)            } //hello()
(7)        }; //class TestClass
(8)    } //test()
(9)
(10)    public static void main(String[] args) {
(11)        TestClass myRV;
(12)        myRV = (new Anonymous()).test();
(13)        myRV.hello();
(14)
(15)        System.out.println( myRV instanceof TestClass);
(16)        System.out.println( myRV.getClass().getName() );
(17)    } //end main()
(18)} //class Anonymous
(19)
(20)class TestClass {
(21)    public void hello() {
(22)        System.out.println("original");
(23)    } //hello()
(24)} //class TestClass

```

Beispiel 30: Anonyme innere Klasse [Anonymous.java](#)

Bildschirmausgabe:

```

$java Anonymous
overridden test!
true
Anonymous$1

```

In der durch die Methode `test` retournierten Ausprägung der Klasse `TestClass` ist die Methode `hello` überschrieben. Da jede innere anonyme Klasse eine bestehende Klasse aus Ausgangsbasis besitzen muß kann die zurückgegebene Ausprägung einer Speicherzelle dieses Typs zugewiesen werden. Entsprechend liefert der [instanceof-Operator](#) auch `true` für den Typtest zurück.

Durch die Redefinition der Methode `hello` wird innerhalb von `main` nicht die ursprüngliche, sondern die überschreibende der anonymen Klasse aufgerufen.

Die letzte Programmzeile in `main` bildet einen Vorgriff auf die noch zu behandelnde [Reflection API](#), welche die Gewinnung von Modellinformationen zur Laufzeit erlaubt. Konkret ermittelt die abgebildete Zeile den Namen der Klasse des in `myRV` gespeicherten Objekts. Java setzt für innere Klassen einen Namen aus der umgebenden Klasse und dem Namen der inneren Klasse, abgetrennt durch das Dollarsymbol, zusammen. Anonyme Klassen werden aufsteigend nummeriert. Daher im Beispiel der Surrogatname `Anonymous$1` für die erste anonyme Klasse innerhalb der Klasse `anonymous`.

Im Grunde genommen handelt es sich bei anonymen inneren Klassen *de facto* um eine besondere Art der [Vererbung](#), welche bereits bekannte Methoden überschreibt. Das überladen existierender Methoden, sowie die Erweiterung der Superklasse um zusätzliche Attribute oder Methoden ist nicht möglich.

Vorwärtsverweis: Ein [reales Beispiel](#) für die Nutzung dieses Sprachmechanismus wird in Kapitel 3 vorgestellt. Im Kontext des in Kapitel 3.2.7 vorgestellten [Abstract Windowing Toolkit \(AWT\)](#) ergeben sich gute reale Anwendungsfälle für anonyme innere Klassen.

Abschlußbemerkung:

Die weiteren Spielarten innerer Klassendefinitionen werden wegen der geringen praktischen Relevanz -- außerhalb des AWT --, und tendenziellen Unübersichtlichkeit des entstehenden Entwurfs nicht diskutiert. Eine gute Referenz findet sich im für Ausbildungszwecke (als HTML) kostenfrei verfügbaren Buch [GoTo Java2](#).

Mit [strictfp](#) existiert seit Java v1.2 ein neues Schlüsselwort zur Modifikation des Klassenverhaltens. Es deaktiviert die erweiterte Gleitpunktdarstellung nach [IEEE 754-1985](#). In diesem Modus werden `float`-Datentypen statt mit 32 mit 43-Bit, bzw. `double`-Datentypen mit 79 statt 64-Bit behandelt.

Durch die bessere Nutzung der vorhandenen Zielhardware ergibt sich neben Performancegewinnen auch eine erhöhte Berechnungsgenauigkeit. Als Resultat kann derselbe Code auf verschiedenen realen Maschinen, je nach Auslegung der Gleitkommaarithmetik, zu verschiedenen Berechnungsergebnissen führen.

Durch Angabe von `strictfp` wird somit wieder portables Verhalten der Fließkommaoperationen erzwungen.

Das Beispiel (nach Kazuyuki SHUDO) zeigt die Verwendung dieses Schlüsselwortes:

```
(1)class StrictfpTest {
(2)    private static double defaultDmul(double a, double b) {
(3)        return a * b;
(4)    }
(5)
(6)    private static strictfp double strictDmul(double a, double b) {
(7)        return a * b;
(8)    }
(9)
(10)   private static double defaultDdiv(double a, double b) {
(11)       return a / b;
(12)   }
(13)
(14)   private static strictfp double strictDdiv(double a, double b) {
(15)       return a / b;
(16)   }
(17)
(18)   public static void main(String[] args) {
(19)       double a, b, c;
(20)
(21)       /* multiplication */
(22)       a = Double.longBitsToDouble(0x0008008000000000L);
(23)       b = Double.longBitsToDouble(0x3ff0000000000001L);
(24)
(25)       System.out.println(a + " (0x0008008000000000)");
(26)       System.out.println(" * " + b + " (0x3ff0000000000001)");
(27)
(28)       c = defaultDmul(a, b);
(29)       System.out.println("default : " + c +
(30)           " (0x" + Long.toHexString(Double.doubleToLongBits(c)) + ")");
(31)
(32)       c = strictDmul(a, b);
(33)       System.out.println("strictfp: " + c +
(34)           " (0x" + Long.toHexString(Double.doubleToLongBits(c)) + ")");
(35)
(36)       System.out.println();
(37)
(38)       /* division */
(39)       a = Double.longBitsToDouble(0x000fffffffffffffffL);
(40)       b = Double.longBitsToDouble(0x3fefffffffffffffffL);
(41)
(42)       System.out.println(a + " (0x000fffffffffffffff)");
(43)       System.out.println(" / " + b + " (0x3fefffffffffffffff)");
(44)   }
```

```

(45)    c = defaultDdiv(a, b);
(46)    System.out.println("default : " + c +
(47)        " (0x" + Long.toHexString(Double.doubleToLongBits(c)) + ")");
(48)
(49)    c = strictDdiv(a, b);
(50)    System.out.println("strictfp: " + c +
(51)        " (0x" + Long.toHexString(Double.doubleToLongBits(c)) + ")");
(52)    }
(53)}

```

Beispiel 31: Verwendung des Schlüsselwortes strictfp [StrictfpTest.java](#)

Die Unterstützung durch die virtuelle Maschine vorausgesetzt, liefert die Ausführung des Programms folgende Ausgabe:

```

1.112808544714844E-308 (0x0008008000000000) * 1.0000000000000002 (0x3ff0000000000001)
default : 1.112808544714844E-308 (0x80080000000000)
strictfp: 1.1128085447148447E-308 (0x800800000000001)

2.225073858507201E-308 (0x000fffffffffffff) / 0.9999999999999999 (0x3fefffffffffffff)
default : 2.2250738585072014E-308 (0x100000000000000)
strictfp: 2.225073858507201E-308 (0xfffffffffffff)

```

SUNs Referenzimplementierung der virtuellen Maschine unterstützen ab Version 1.3 auf den Intelplattformen Windows und Linux die korrekte Umsetzung des erweiterten Fließkommaformats.

Objekterzeugung:

Die Erzeugung konkreter Ausprägungen (auch: Instanzen oder Objekte) einer Klasse geschieht üblicherweise durch den new-Operator.

Die Syntax lautet: *Type Variable = new Type(Parameters)*

Wird ein Objekt nicht mehr durch den Programmierer referenziert, so wird es durch den Garbage Collector aus dem Speicher entfernt. Dies kann u. U. auch erst verzögert geschehen, da der Garbage Collector als eigener asynchroner Thread der virtuellen Maschine realisiert ist. (Der Aufruf [System.gc\(\)](#) schlägt der virtuellen Maschine vor den Garbage Collector bei Gelegenheit aufzurufen.)

[Weiterführende Informationen zum Thema innere Klassen.](#)

▲ 2.4.3 Attribute

Attribute stellen in der objektorientierten Programmierung den üblichen -- und bei strenger Betrachtung den einzigen -- Weg zur Ablage dynamischer Zustandsinformation eines Objekts dar. (Selbstverständlich können auch Variablen innerhalb von [Methoden](#) beliebige Informationen aufnehmen. Jedoch sind diese Inhalte im Allgemeinen nach Verlassen des Gültigkeitsbereichs verloren).

[Abbildung 2](#) stellt die Platzierung der Attribute in UML-Notation dar.

Vereinfachte **Syntax** einer Attributdefinition:

```

(public,
protected,
private,
static,
final,
transient,
volatile) Identifizier []optional (= Initialization)optional
zero or more

```

Die Syntaxkomponenten im Einzelnen:

Syntaxelement Semantik

public	Das Attribut ist allgemein für alle anderen Klassen sichtbar.
protected	Das Attribut ist ausschließlich für die definierende Klasse sowie diejenigen sichtbar die von aktuellen erben, sowie alle Klassen desselben Packages.
private	Das Attribut ist nur innerhalb der deklarierenden Klasse sichtbar.
static	Gültigkeitsbereich des Attributs ist die gesamte Klasse. D.h. alle Objekte dieser Klasse teilen sich <i>dasselbe</i> Attribut (= derselbe Speicherplatz). Ein so deklariertes Attribut hat in allen Ausprägungen denselben Wert; Änderungen finden automatisch synchronisiert in allen Objekten statt. Ohne Angabe dieses Schlüsselwortes ist der Scope auf die Objektebene festgelegt.
final	Das Attribut ist konstant, und kann nach seiner (zwingend anzugebenden!) Initialisierung nicht mehr verändert werden.

transient	So ausgezeichnete Attribute sind nicht Teil des persistenten Objektzustandes, und werden bei einer Ablage auf Sekundärspeicher (File, Datenbank, etc.) ignoriert.
volatile	Kennzeichnet ein Attribut als bei Optimierungen nicht zu berücksichtigen. Die Semantik und Anwendung ist ähnlich zur aus C/C++ bekannten Mimik.

```

(1)public class Attributes {
(2)    int            testAttribute1;
(3)    public short   testAttribute2 = 99;
(4)    private long    testAttribute3 = 5L;
(5)    protected boolean testAttribute4;
(6)    volatile private char testAttribute5;
(7)    public transient long testAttribute6 = 0x42;
(8)
(9)    static final double pi = 3.141592654;
(10)
(11)    TestClass testAttributeC1;
(12)    TestClass testAttributeC2 = new TestClass();
(13)} //class Attributes
(14)
(15)class TestClass {
(16)} //class TestClass

```

Beispiel 32: Einige Attributdefinitionen [Attributes.java](#)

Abschlußbemerkungen:

- Aus Gründen der Speicherplatzgründen empfiehlt sich die Definition von Konstanten (wie pi im Beispiel) als `static final`.
- Java erlaubt, anders als C++, keinerlei Datenstrukturen außerhalb von Klassen. Globale Variablen, sowie wiederverwendbare „Allzweckvariablen“ -- beispielsweise als Schleifenzähler -- können daher nicht in der gewohnten (Un-)Sitte realisiert werden.

▲ 2.4.4 Operationen und Methoden

Operationen und Methoden bilden das dynamische Verhalten eines Objekts ab. Während der Begriff *Operation* nur die Signatur, bestehend aus Rückgabebetyp, Operationsnamen und der Parameterliste, bezeichnet, deckt *Methode* auch die programmiersprachliche Umsetzung ab.

Java erlaubt ausschließlich die Definition von Methoden innerhalb von Klassen. Globale Funktionen sind ebenso wie globale Variablen nicht möglich!

Vereinfachte **Syntax** einer Operation:

```

(public,
protected,
private,
abstract,
static,           ResultType      Identifier      (ParameterList)      (throws ExceptionList)optional
final,
synchronized,
native,
strictfp)zero or more

```

Die Syntaxkomponenten im Einzelnen:

Syntaxelement Semantik

public	Die Methode ist allgemein für alle anderen Klassen sichtbar.
protected	Die Methode ist ausschließlich für Klassen sichtbar die von der aktuellen erben, sowie alle Klassen desselben Packages.
private	Die Methode ist nur innerhalb der deklarierenden Klasse sichtbar.
abstract	Definiert analog zum Schlüsselwort auf Klassenebene, daß eine so gekennzeichnete Operation keine Implementierung bereitstellt; mithin keine Methode implementiert.
static	Gültigkeitsbereich dieser Methode ist die Klasse. In der Anwendung bedeutet dies, daß eine so deklarierte Methode sowohl auf der Klasse selbst, als auch einem Objekt dieser Klasse aufgerufen werden kann. Beispiel
final	Die Methode darf nicht in einer erbenden Klasse überschrieben werden.
synchronized	Auf einer so deklarierten Methode wird durch das Laufzeitsysteme ein wechselseitiger Ausschuß realisiert. Das bedeutet, daß sich nur jeweils ein Thread innerhalb einer Methode des Objektes befinden darf.

native	Eine so gekennzeichnete Methode ist nicht in Java realisiert, sondern wird als nativer Code einer anderen Programmiersprache (z.B. C/C++) realisiert. Das JDK bietet für die beiden genannten Sprachen Unterstützung in Form automatisch generierter Headerdateien an.
strictfp	Deaktiviert Nutzung plattformspezifischer Gleitkommahardware. analog strictfp auf Klassenebene

Aufgrund der strengen Typisierung der Programmiersprache ist jede Javaoperation über ihren Rückgabewert typisiert. Als **Rückgabetypen** stehen alle [primitiven built-in Datentypen](#), alle [Klassen](#) und [Schnittstellen](#), sowie der explizite *Nichttyp* void zur Verfügung.

Wie aus C/C++ bekannt geben void-Methoden keine Werte an den Aufrufer zurück, und können daher nicht innerhalb von Ausdrücken verwendet werden.

Ebenso analog der bekannten Mimik wird die explizite Rückgabe eines Wertes durch das Schlüsselwort return eingeleitet. Wie in (neuen) C/C++-Übersetzern üblich, prüft auch Java die Existenz einer erreichbaren typkompatiblen Return-Anweisung innerhalb des Methodenrumpfes.

Die auf den Operationsnamen folgende **Parameterliste** setzt sich aus einer Folge von Typen und Parameternamen zusammen, kann jedoch auch leer sein.

Verursacht durch das Fehlen expliziter Referenztypen (wie Zeiger) werden alle Parameter, die [primitive build-in Typen](#) sind **by value** und alle objektartigen Parameter per Vorgabe **by reference** übergeben.

Hinweis: Es existiert kein Mechanismus dieses Verhalten zu überschreiben oder abzuändern! Lediglich für die by reference Interpretation der Primitivtypen ist mit den [Wrapper Typen](#) eine Standardmethodik vorgesehen.

Genaugenommen kommt auch für Objekte, die als Parameter einer Methode verwendet werden, eine Wertübergabe zum Einsatz. Jedoch wird in diesem Falle nicht der Wert des Objektes übergeben, d.h. es wird keine Kopie des Objektes erzeugt und an die aufzurufende Methode übergeben. Vielmehr wird eine Kopie der Referenz auf das Objekt übergeben.

Dieser Umstand führt dazu, daß auch objektwertige Parameter innerhalb eines Methodenrumpfes nicht direkt modifiziert werden können, sondern hierfür auf Methoden des zu verändernden Objekts zurückgegriffen werden muß. [Mehr Informationen hierzu.](#)

Die **Signatur einer Javamethode** wird aus Operationsname und der Parameterliste, unter Berücksichtigung der Parameterreihenfolge, gebildet.

Sichtbarkeitsattribute, ebenso wie der Rückgabetyt und weitere Eigenschaften gehen nicht in die Signaturbildung ein.

Innerhalb des **Methodenrumpfs** sind alle aus 2.3 bekannten [Kontrollstrukturen](#) zugelassen. Darüberhinaus kann an beliebiger Stelle die Deklaration lokaler Variablen erfolgen. Im Gegensatz zu C/C++ ist die Lebensdauer lokaler Variablen strikt an die des umgebenden Blocks gebunden. Daher steht das Schlüsselwort static für die Variablendefinition innerhalb von Methoden nicht zur Verfügung.

Der Aufruf von Methoden erfolgt in der bekannten *Punktnotation*, die in allgemeiner Form als `Objekt.Methode (Parameterliste)` oder entsprechend `Klasse.Methode(Parameterliste)` bei Klassenmethoden beschrieben werden kann.

Eine Sonderrolle spielt das **Schlüsselwort this im Methodenrumpf**. Es steht als impliziter Parameter in allen nicht-statischen Methoden zur Verfügung. `this` referenziert immer das aktuelle Objekt. Die lokale Variable `this` ist dabei immer von Typ [final](#) und erlaubt keine Neuzuweisungen.

Üblicherweise ist die Verwendung von `this` nicht explizit erforderlich. Zur Behebung von Namenskonflikten zwischen Übergabeparametern und lokalen Variablen leistet es jedoch wertvolle Dienste. Weitere Anwendung findet das Schlüsselwort zur [Aufruf von Konstruktoren](#).

```
(1)public class ThisDemo {
(2)    private int i = 42;
(3)
(4)    public void setI(int i) {
(5)        this.i = i;
(6)    } //setI()
(7)
(8)    public int getI() {
(9)        return this.i;
(10)    } //getI()
(11)
(12)    public static void main(String[] args) {
(13)        ThisDemo td = new ThisDemo();
(14)        System.out.println("value of i="+td.getI() );
(15)        td.setI(45);
(16)        System.out.println("value of i="+td.getI() );
(17)    } //main()
(18)} //class ThisDemo
```

Beispiel 33: Zugriff auf Objekteigenschaften mit `this` [ThisDemo.java](#)

Bildschirmausgabe:

```
$java ThisDemo
value of i=42
value of i=45
```


Das Beispiel zeigt die Verwendung der Eigenobjektreferenz `this` innerhalb der Methoden `setI` und `getI`. Während die Referenzierung von `i` innerhalb `getI` auch ohne vorstelltes explizites `this` eindeutig auflösbar ist, muß in `setI` das Schlüsselwort zwingend angegeben werden um dem Namenskonflikt zwischen dem Parameter `i` und dem gleichnamigen Attribut aufzulösen.

Analog zu den Klassenattributen werden durch das Schlüsselwort **static** **Klassenmethoden** definiert. Auf sie kann unabhängig von der Existenz konkreter Objekte der Klasse zugegriffen werden. Aufgrund ihrer instanzenunabhängigen Natur besitzen statische Methoden keine `this` Referenz, da ein so zu referenzierendes Objekt nicht existiert.

```
(1)public class StaticMethodTest {
(2)    public static void helloWorld() {
(3)        System.out.println("hello world");
(4)    } //helloWorld()
(5)
(6)    public static void main(String[] args) {
(7)        StaticMethodTest.helloWorld();
(8)        (new StaticMethodTest()).helloWorld();
(9)    } //main()
(10)} //class StaticMethodTest
```

Beispiel 34: Statische Methoden [StaticMethodTest.java](#)

Besondere Methoden:

Konstruktoren

Identisch benannt zur beherbergenden Klasse

Syntaktisch ähneln die Konstruktoren den „normalen“ Operationsdeklarationen. Jedoch mit dem Unterschied, daß kein Rückgabetypp spezifiziert werden kann.

Der Konstruktorenaufruf wird automatisch durch den Übersetzer plaziert.

Existiert kein expliziter Konstruktor, so wird während des Übersetzungsvorganges ein unparametrisierter Vorgabekonstruktor erzeugt. Dieser enthält keinerlei eigene Funktionalität, sondern ruft nur den entsprechenden parameterlosen Konstruktor der Superklasse auf. Verfügt eine Klasse hingegen ausschließlich über parametrisierte Konstruktoren, so wird kein unparametrisierter Defaultkonstruktor angelegt.

Aufrufe unter den verschiedenen Konstruktoren können durch das bekannte **Schlüsselwort** **this** vorgenommen werden. Hierbei werden die einzelnen Konstruktoren wie normale Methoden behandelt. Der Aufruf erfolgt durch `this` gefolgt von den Konstruktorenparametern in runden Klammern. Der kaskadierende Konstruktorenaufruf muß zwingend als erstes Statement des Anweisungsblockes plaziert werden.

Zwar verfügt die Konstruktormethode über keinen expliziten Rückgabetypp, wird aber intern als `void`-Typisiert behandelt.

```
(1)public class Construct {
(2)    public Construct() {
(3)        System.out.println("object created using explicitly stated parameterless
Constructur");
(4)    } //constructor
(5)
(6)    public Construct(int i) {
(7)        System.out.println("object Constructed by parametrized Constructor, parameter
i="+i);
(8)    } //constructor
(9)
(10)    public static void main(String args[]) {
(11)        new Construct();
(12)        new Construct(1);
(13)        new TestClass1();
(14)    } //main()
(15)} //class Construct
(16)
(17)class TestClass1 extends TestClass2 {
(18)    //nothing!
(19)} //TestClass1
(20)
(21)class TestClass2 {
(22)    TestClass2() {
(23)        this(1);
(24)        System.out.println("Constructor of TestClass2 called");
(25)    } //constructor
(26)
(27)    TestClass2(int i) {
(28)        this ((short) 2, (byte) 3);
(29)        System.out.println("Constructor of TestClass2 called paramter i="+i);
(30)    } //constructor
(31)
(32)    TestClass2(short s, byte b) {
(33)        System.out.println("Constructor of TestClass2 called paramter s="+s+" parameter
b="+b);
(34)    } //constructor
(35)} //class TestClass2
```

Beispiel 35: Verschiedene Konstruktoren [Construct.java](#)**Nicht-öffentliche Konstruktoren und Fabriken:**

Häufig anzutreffen sind Klassen, die zwar einen explizit definierten Konstruktor bieten, diesen jedoch `private` deklarieren. In der Konsequenz ist eine Objekterzeugung per `new` nicht möglich.

Zumeist wird dieser Ansatz angewandt, wenn die Objekterzeugung aufwendig ist und nicht dem Anwender überlassen werden kann oder soll.

Um dennoch Objekte der betreffenden Klasse erzeugen zu können wird eine statische *Fabrik*-Methode eingeführt, welche die Objekterzeugung übernimmt und implizit `new` aufruft.

```
(1)public class Construct2 {
(2)    private Construct2() {
(3)        System.out.println("hello world");
(4)    } //constructor
(5)
(6)    public static void main(String[] args) {
(7)        new Construct2();
(8)        /* new OtherClass(); would fail since constructor is declared to be private */
(9)        OtherClass oc = OtherClass.OtherClassFactory(); //object creation using simple
factory approach
(10)    } //main()
(11)} //class Construct2
(12)
(13)class OtherClass {
(14)    public static OtherClass OtherClassFactory() {
(15)        return new OtherClass();
(16)    } //constructor
(17)
(18)    private OtherClass() {
(19)        System.out.println("other class created");
(20)    } //constructor
(21)} //class OtherClass
```

Beispiel 36: Nicht-öffentliche Konstruktoren und Fabrikmethoden zur Objekterzeugung [Construct2.java](#)**Bildschirmausgabe:**

```
$java Construct2
hello world
other class created
```

Der Konstruktorenaufruf innerhalb der Klasse `construct2` kann bei Absetzen des `new` ausgeführt werden, da aus der Klasse heraus (konkret: innerhalb einer der statischen Methode `main`) ein Objekt dieser Klasse erzeugt wird -- der `private` Konstruktor ist somit zugreifbar.

Hingegen ist die Erzeugung eines Objekts von `otherClass` per `new` nicht möglich, da der dort definierte `private` Konstruktor nicht aus `construct2` heraus referenzierbar ist. Der entsprechende Fehler wird bereits zum Übersetzungszeitpunkt erkannt.

Die Erzeugung von Objekten der Klasse `otherClass` ist ausschließlich über die statische Methode `otherClassFactory` möglich. Sie ruft intern den privaten Konstruktor auf, der an dieser Stelle sicht- und zugreifbar ist.

(siehe [Java API Specification](#), [siehe Java Language Specification](#))

Ein Beispiel für eine Klasse, die weder über eine Factorymethode, noch über öffentliche Konstruktoren verfügt ist die Standard-API-Klasse [Void](#).

Ihre Implementierung ist in der API wie folgt festgelegt:

```
public final
class Void {
    public static final Class TYPE = Class.getPrimitiveClass("void");

    private Void() {}
}
```

Noch vor dem Konstruktor werden die **statischen Initialisierungen** aufgerufen. Aufgrund der Reihenfolge in der Methodenabarbeitung existiert zum Ablaufzeitpunkt der statischen Initialisierungen das zu erzeugende Objekt noch nicht; der `static`-Block wird quasi zum Ladezeitpunkt der Klasse ausgeführt. Daher können zu diesem Zeitpunkt keine Zugriffe auf nichtstatische Speicherobjekte (Methoden und Attribute) erfolgen. Zugriffe auf statische Attribute und Methoden sind jedoch möglich.

Der Initialisierungsblock darf nicht durch Unterbrechungsanweisungen wie `break` oder `return` vorzeitig verlassen werden. Hierunter fallen auch Exceptions, die zum vorzeitigen Verlassen des Anweisungsblockes führen würden.

```

(1)public class StaticInit {
(2)    static int i=42;
(3)
(4)    static {
(5)        System.out.println("value of i="+i);
(6)        helloWorld();
(7)        i = 43;
(8)    } //static
(9)
(10)   public static void helloWorld() {
(11)       System.out.println("hello world");
(12)   } //helloWorld()
(13)
(14)   public static void main(String[] args) {
(15)       StaticInit myObj = new StaticInit();
(16)   } //main()
(17)
(18)   public StaticInit() {
(19)       System.out.println("value of i="+i);
(20)   } //constructor
(21)} //class StaticInit

```

Beispiel 37: Statische Initialisierung [StaticInit.java](#)

BildschirmAusgabe:

```

$java StaticInit
value of i=42
hello world
value of i=43

```

Das Beispiel zeigt zunächst den Zugriff auf das statische Attribut (Klassenattribut) `i` direkt nach dessen Definition und Initialisierung. Zu diesem Zeitpunkt ist der Initialisierungswert 42 gesetzt. Der Aufruf von `helloWorld` demonstriert die Möglichkeit vor der Objekterzeugung statische Methoden auszuführen. Nach Abarbeitung der statischen Initialisierung wird der Konstruktor bearbeitet. In ihm steht der Wert von `i` nur noch in der durch die statische Initialisierung modifizierten Form zur Verfügung.

Ausführungsreihenfolge der konstruierenden Codesequenzen:

(als Pseudocode):

```

for-each (Superclass s) {
    s.AttributeInitialization()
    s.StaticBlock()
} //for-each
for-each (Superclass s) {
    s.Constructor()
}

```

Beginnend mit der hierachiehöchsten Superklasse werden zunächst die Initialisierungen der Attribute, im Anschluß daran (falls vorhanden) der static-Block dieser Klasse, abgearbeitet.

In einem zweiten Durchlauf über die Klassenhierarchie werden die Konstruktoren der Superklassen in derselben Reihenfolge zur Ausführung gebracht.

```

(1)public class SuperClassConstruct extends Class2 {
(2)    static int i=6;
(3)
(4)    static {
(5)        System.out.println("i="+i--);
(6)        System.out.println("static initialization of class SuperClassConstruct");
(7)        System.out.println("i="+i);
(8)    } //static
(9)
(10)   public SuperClassConstruct() {
(11)       System.out.println("object of class SuperClassConstruct constructed");
(12)   } //constructor
(13)
(14)   public static void main(String[] args) {
(15)       new SuperClassConstruct();
(16)   } //main()
(17)} //class SuperClassConstruct
(18)
(19)class Class2 extends Class1 {
(20)    static int i=4;
(21)
(22)    static {
(23)        System.out.println("i="+i--);
(24)        System.out.println("static initialization of class Class2");
(25)        System.out.println("i="+i);
(26)    } //static

```

```

(27)
(28)     public Class2() {
(29)         System.out.println("object of class Class2 constructed");
(30)     } //constructor
(31)} //class Class2
(32)
(33)class Class1 {
(34)     static int i=2;
(35)     static {
(36)         System.out.println("i="+i--);
(37)         System.out.println("static initialization of class Class1");
(38)         System.out.println("i="+i);
(39)     } //static
(40)
(41)     public Class1() {
(42)         System.out.println("object of class Class1 constructed");
(43)     } //constructor
(44)} //class Class2

```

Beispiel 38: Initialisierungsreihenfolge [SuperClassConstruct.java](#)

BildschirmAusgabe:

```

$java SuperClassConstruct
i=2
static initialization of class class1
i=1
i=4
static initialization of class class2
i=3
i=6
static initialization of class superClassConstruct
i=5
object of class class1 constructed
object of class class2 constructed
object of class superClassConstruct constructed

```

Destruktoren --finalize

Zwar können in Java, anders als in C++, Objekte nicht durch Destruktorenaufruf zerstört werden, sondern nur durch Null-Zuweisung als nicht mehr benötigt markiert werden, Destruktoren werden jedoch weiterhin explizit zur Verfügung gestellt.

Destruktoren werden nicht durch den Anwender aufgerufen, sondern implizit erst zum Zerstörungszeitpunkt eines Objekts. Genaugenommen ist die Ausführung des Destruktors nicht garantiert. Terminiert die Applikation vor dem Ablauf des [Garbage Collectors](#), so werden die Objekte implizit durch das Betriebssystem aus dem Speicher entfernt, ohne das die Java-Laufzeitumgebung zunächst die finalize-Methoden aufruft.

Die Sichtbarkeitseinschränkung von finalize muß mindestens protected sein. Dies rührt von der impliziten Überschreibung der von Object ererbten Methode finalize her. (vgl. [java.lang.Object:finalize](#))

```

(1)public class DestruktorTest {
(2)     public static void main(String[] args) {
(3)         Test t1 = new Test();
(4)         t1 = null;
(5)         System.gc();
(6)     } //main()
(7)} //class DestruktorTest
(8)
(9)class Test {
(10)    public void finalize() {
(11)        System.out.println("destructor of class Test called");
(12)    } //finalize()
(13)} //class Test

```

Beispiel 39: Destruktorenaufruf durch Garbage Collector [DestruktorTest.java](#)

Im Beispiel wird das Objekt t1 zunächst durch Nullsetzung zum Löschen markiert, und im anschließenden Garbage Collector-Aufruf gelöscht. Während des Entfernens aus dem Speicher wird die dargestellte Nachricht am Bildschirm ausgegeben.

Hauptmethode -- main

Pro Java-Applikation kann maximal eine Methode der Signatur main(String[]) existieren. Sie wird beim Startvorgang innerhalb der öffentlichen Klasse gesucht, die namensgebend für die Klassendatei ist.

Applets verfügen hingegen über keine automatisch aufgerufene main-Methode, sondern werden durch init() initialisiert und durch das im Anschluß darauf aufgerufene start() ausgeführt.

Die Methode toString

Ist auf der Superklasse [Object](#) aller Javaklassen definiert. Sie wird von allen Klassen der Standard-API implementiert. Durch sie wird immer eine Zeichenkettenrepräsentation des aktuellen Objekts zurückgegeben. Diese Methode wird standardmäßig bei der Ausgabe per System.out.println aufgerufen.

Zum Abschluß: Sichtbarkeit von Attributen und Methoden im Überblick:

	definierende Klasse	Abgeleitete Klasse	Package	Alle anderen
<i>default</i> (keine explizite Festlegung)	✓			
public	✓	✓	✓	✓
private	✓			
protected	✓	✓ Zugriff auf Attribute und Methoden der Subklasse	✓	

Anmerkung: Der Zugriff auf als protected deklarierte Attribute und Methoden ist auch über Objektreferenzen möglich, die denselben Typ haben wie die definierende Klasse.

▲ 2.4.5 Aufzählungstypen

Ab Version 1.5 führt Java mit dem Schlüsselwort `enum` einen eigenständigen und expliziten Mechanismus zur Definition von Aufzählungstypen ein.

Konzeptionell sind Aufzählungstypen identisch zu Variablen, die innerhalb des Rumpfes einer Methode deklarierten werden oder Variablen und Attributen einer Klasse gleichgestellt. Aus diesem Grunde ähnelt die Definitionssyntax auch den bekannten Darstellungsformen:

Vereinfachte **Syntax** der Enum-Definition:

```
(public,
protected,
private) one of      staticopt      finalopt      enum      Identifier      {value}
```

Im einfachsten Anwendungsfall besteht die Definition eines Aufzählungstypen aus der durch Kommata voneinander separierten vollständigen Aufzählung aller Werte.

Das Beispiel zeigt die Bildung des Aufzählungstyps `season`, der durch die zulässigen Werte `winter`, `spring`, `summer` und `fall` konstituiert wird.

```
(1) public class EnumTest1 {
(2)     public static void main(String[] args) {
(3)         enum season { winter, spring, summer, fall; }
(4)
(5)         season s1 = season.winter;
(6)
(7)         if (s1 == season.winter)
(8)             System.out.println("It's " + s1);
(9)
(10)        season s2 = season.winter;
(11)    } //main()
(12)} //class EnumTest1
```

Beispiel 40: Definition eines einfachen Aufzählungstyps [EnumTest1.java](#)

Die Verwendung von Aufzählungstypen entspricht der von primitiven Typen. Aus diesem Grund ist keine Anforderung von Speicherplatz durch das `new`-Schlüsselwort notwendig. Der Übersetzer verhindert sogar aktiv die Instanziierung eines Aufzählungstyps via `new` und bricht beim Versuch mit der Fehlermeldung `enum types may not be instantiated` ab.

Im Gegensatz zu Ausprägungen der Primitivtypen besitzen Aufzählungsinstanzen jedoch keinen Vorgabewert mit dem sie standardmäßig initialisiert werden. Stattdessen führt die Verwendung einer nichtinitialisierten Ausprägung eines Aufzählungstypen zu einem Übersetzungsfehler (Fehlermeldung: `variable ... might not have been initialized`).

Ansonsten bietet die Umsetzung die für Primitivtypen bekannten Eigenschaften. So liefert die Ausgabe diejenige Zeichenkette (d.h. den Wert) mit dem Aufzählungsinstanz belegt wurde.

Ebenfalls identisch zu den Primitivtypen besitzen Aufzählungsinstanzen keine Identität. Dies äußert sich darin, daß zwei mit demselben Wert belegte Aufzählungen als identisch betrachtet werden.

Nicht identisch sind hingegen Ausprägungen verschiedener Aufzählungstypen, die vermeintlich denselben Wert enthalten, d.h. in deren zur Definition verwendeten Werteliste sich lexikalisch dieselben Einträge finden.

So würde die nachfolgende Zuweisung bereits durch den Übersetzer (mit der Fehlermeldung `incompatible types`)

abgelehnt

```
enum seasonE { winter, spring, summer, fall; };
enum seasonG { winter, fröhling, sommer, herbst; };
seasonE s = seasonG.winter;
```

Dasselbe gilt auch für den Versuch des Vergleichs der Inhalte zweiter Aufzählungsausprägungen, wie sie durch das nachstehende Codefragment versucht wird.

```
enum seasonE { winter, spring, summer, fall; };
enum seasonG { winter, fröhling, sommer, herbst; };
seasonE s1 = seasonE.winter;
seasonG s2 = seasonG.winter;
if (s1==s2) ...
```

Auch in diesem Fall wird bereits zum Übersetzungszeitpunkt durch die Fehlermeldung `incomparable types: seasonG and seasonE` auf den Fehler hingewiesen.

In Erweiterung der bisher vorgestellten Syntax lassen sich Aufzählungstypen sogar „klassenartig“ ausbauen. Diese Erweiterung erlaubt es die Elemente des Aufzählungstypen wahlfrei an selbstdefinierte Eigenschaften zu binden. Konzeptionell werden diese Eigenschaften dabei als Attribute des Aufzählungstypen aufgefaßt, die durch einen durch den Programmierer bereitzustellenden Konstruktor zugewiesen werden. Der Konstruktoreaufruf erfolgt dabei automatisch durch das Laufzeitsystem zum Definitionszeitpunkt eines Aufzählungstypen für alle konstituierenden Inhaltselemente. Das nachfolgende Beispiel zeigt eine Verwendung des erweiterten Konzepts:

```
(1)public class EnumTest2 {
(2)    public enum Coin {
(3)        penny(1), nickel(5), dime(10), quarter(25);
(4)        private int value;
(5)        Coin(int value) {
(6)            System.out.println("creating: "+value);
(7)            this.value = value;
(8)        } //constructor
(9)        public int value() {
(10)            return value;
(11)        } //value()
(12)    } //enum Coin
(13)
(14)    public static void main(String[] args) {
(15)        Coin c1 = Coin.dime;
(16)        System.out.println( c1.value() );
(17)    } //main
(18)} //class EnumTest2
```

Beispiel 41: Definition eines klassenartigen Aufzählungstyps [EnumTest2.java](#)

Das Beispiel definiert den Typ `Coin` mit seinen Inhaltstypen `penny`, `nickel`, `dime` und `quarter`. Den Inhaltstypen wird durch Konstruktoreaufruf jeweils ein Inhaltswert zugeordnet. Dieser Wert wird der definierten privaten Variable `value` zugewiesen.

Durch die Methode `value` kann der dem jeweiligen Inhaltstyp zugewiesene Wert ausgelesen werden. Methoden, die den Inhalt der definierten Variable schreiben können zwar definiert werden, jedoch werden Wertänderungen nicht auf die vordefinierten Inhaltstypen synchronisiert.

Die Anzahl der intern mit einem Wert verbundenen Festwerte ist hier bei nicht beschränkt. Das abschließende Beispiel zeigt die Zuweisung von zwei Einzelwerten im Konstruktor:

```
(1)public class EnumTest3 {
(2)    public enum Time {
(3)        highNoon(12,00), sunSet(20,30), sunRise(5,30), teaTime(17,00);
(4)        private int hour;
(5)        private int minute;
(6)
(7)        Time(int hour, int minute) {
(8)            this.hour = hour;
(9)            this.minute = minute;
(10)        } //constructor
(11)        public void printTime() {
(12)            System.out.print(hour+": "+minute);
(13)        } //printTime()
(14)    } //enum Time
(15)
(16)    public static void main(String[] args) {
(17)        Time t1 = Time.sunSet;
(18)        System.out.print( "Sunset is at: ");
(19)        t1.printTime();
(20)    } //end main
(21)} //class EnumTest3
```

▲ 2.4.6 Wrapper-Typen

Korrespondierend zu jedem [primitiven Datentypen](#) in Java gibt es einen *Wrapper Typen*.

Sie kapselt den zugrundelegenden Primitivtyp in einer eigenen Klasse, und stellt einige Servicemethoden bereit.

Objekte aller Wrappertypklassen können nur bei ihrer Erzeugung mit Werten versehen werden, die über die gesamte Lebensdauer nicht mehr verändert werden können.

Primitivtyp

[boolean](#)

[byte](#)

[short](#)

[int](#)

[long](#)

[float](#)

[double](#)

[void](#)

Dieser Typ ist nicht als Datentyp für Variablen und Attribute verfügbar, sondern kann nur als Rückgabtyp von Operationen spezifiziert werden.

Wrapper Typ

[Boolean](#)

[Byte](#)

[Short](#)

[Integer](#)

[Long](#)

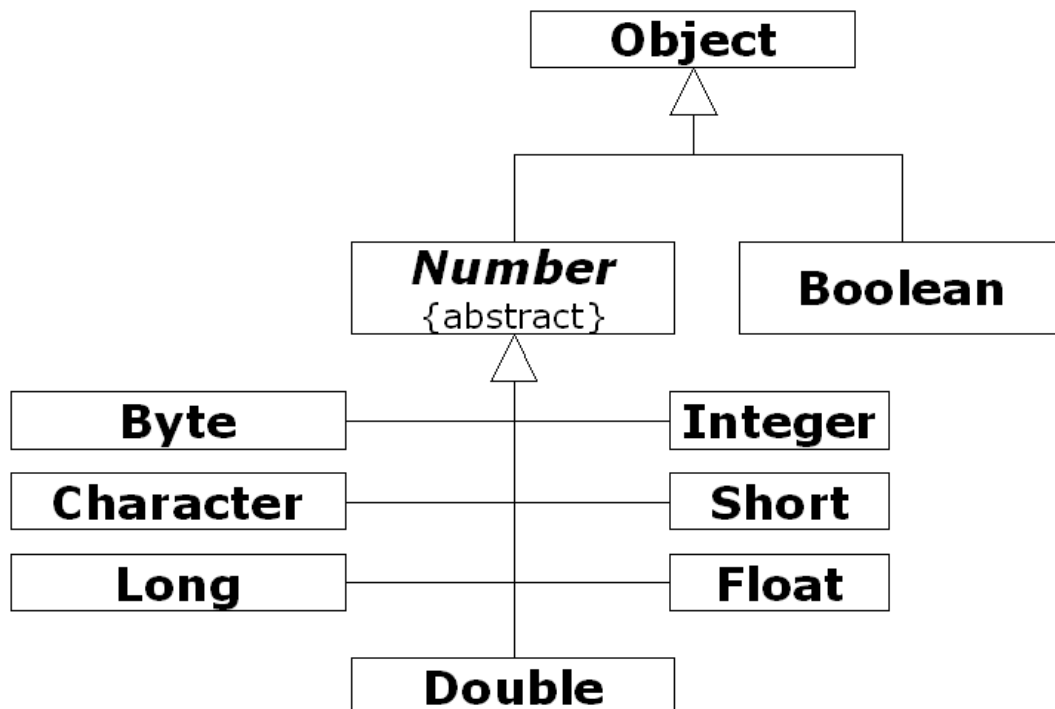
[Float](#)

[Double](#)

[Void](#)

Kann nicht instanziiert werden.

Die Abbildung 9 zeigt die Organisation der Wrapperklassen innerhalb der Standard-API im Überblick:



```

(1)public class CBRCBV {
(2)    public static void main(String[] args) {
(3)        TestClass testObj = new TestClass();
(4)        int testVar;
(5)        Byte testWrapperType;
(6)
(7)        testObj.setS((short) 42);
(8)        testWrapperType = new Byte((byte) 12);
(9)        testVar = 50;
(10)
(11)        System.out.println("values before method run");
(12)        System.out.println("testVar = "+testVar);
(13)        System.out.println("testWrapperType = "+testWrapperType.byteValue() );
(14)        System.out.println("testObj.s = "+testObj.getS() );
(15)
(16)        aMethod(testVar, testWrapperType, testObj);
(17)
(18)        System.out.println("values after method run");
(19)        System.out.println("testVar = "+testVar);
  
```

```

(20)         System.out.println("testWrapperType = "+testWrapperType.byteValue() );
(21)         System.out.println("testObj.s = "+testObj.getS() );
(22)     } //main()
(23)
(24)     private static void aMethod(int i, Byte b, TestClass tc)      {
(25)         i++;
(26)         b = new Byte((byte) 0);
(27)         tc.setS( (short) (tc.getS()+1) );
(28)
(29)         System.out.println("values within method after modification:");
(30)         System.out.println("testVar = "+i);
(31)         System.out.println("testWrapperType = "+b.byteValue() );
(32)         System.out.println("testObj.s = "+tc.getS() );
(33)     } //aMethod()
(34)} //class CBRCBV
(35)
(36)class TestClass {
(37)    private short s;
(38)
(39)    public short getS() {
(40)        return s;
(41)    } //getS
(42)
(43)    public void setS(short s) {
(44)        this.s = s;
(45)    } //setS()
(46)} //TestClass

```

Beispiel 43: Verwendung von primitiven, Wrapper und objektwertigen Typen [CBRCBV.java](#)

Bildschirmausgabe:

```

values before method run
testVar = 50
testWrapperType = 12
testObj.s = 42
values within method after modification:
testVar = 51
testWrapperType = 0
testObj.s = 43
values after method run
testVar = 50
testWrapperType = 12
testObj.s = 43

```

Im Beispiel werden zunächst Exemplare der drei verschiedenen Typfamilien -- Primitivtyp, Wrapper Typ und Objekt -- erzeugt und mit Werten versehen.

Innerhalb der Methode `aMethod`, die alle drei Variablen als Übergabeparameter erhält, werden die Inhalte lokal geändert. Während die `int`-Variable direkt beschrieben werden kann, wird die Änderung des objektwertigen Parameters `tc` durch eine Methode der Klasse `TestClass` realisiert. Auf dem Wrapper Typen ist durch die Java-API keine Modifikationsroutine vorgesehen. Daher wird der übergebenen Referenz ein neues Objekt zugewiesen. Nach Ausführung der Methode `aMethod` werden die Werte nochmals ausgegeben. Hier zeigt sich, daß auch für Objekte **kein echtes Call-by-Reference** zur Verfügung steht, sondern lediglich eine Kopie auf die Referenz übergeben wurde. Aus diesem Grunde sind die Änderungen sowohl am übergebenen Primitivtypen als auch an der Variable des Typs `Byte` verloren.

Als einzige Möglichkeit zur Realisierung von Modifikationen an einem Objekt erweisen sich die Methoden dieses Objekts.

Mehr Information hierzu:

- [Understanding that parameters are passed by value and not by reference](#)
- [Pass-by-value semantics in Java applications](#)

Alle Wrappertypen bieten bestimmte Servicemethoden an, die in der Praxis häufig auftretende Anforderungen erfüllen. Darunter fallen neben der Ausgabe des Wrappertypeninhaltes (d.h. des gekapselten Primitivwertes) auch Umsetzungen der auf Objekten nicht zur Verfügung stehenden Typumwandlungen (casts), sowie Methoden zur Erzeugung der Primitivrepräsentationen aus anderen als den Wrappertypendarstellungen (z.B. aus beliebigen Zeichenketten).

Wichtige und gebräuchliche Servicemethoden, sowie weitere Charakteristika der Wrappertypen:

- Alle Wrapperklassen sind `final` deklariert, und können daher nicht weiter spezialisiert werden.
- ...**Value**

Beispiele: [booleanValue](#), [byteValue](#), [shortValue](#), [intValue](#), [longValue](#), [floatValue](#), [doubleValue](#)

Die abstrakte Signatur lautet in allen Fällen: `public T TValue()`, wobei `T` der jeweilige Primitivtyp ist (entsprechend für den Wrappertypen um `int`: `public int intValue()`).

Gibt den intern gekapselten Wert unverändert zurück.

Darüberhinaus existieren Methoden zur Ausgabe eines numerischen Wrappertypen in einen beliebigen [numerischen Primitivtypen](#). Diese ersetzen somit die explizite Typumwandlung.

- **Konstruktoren**

Erlauben die Erzeugung des Wrappertypen aus einem Ausdruck vom Typ des gekapselten Typen oder aus einem String. (Beispiele: `Boolean (boolean value)`, `Boolean (String s)`, `Integer (int value)`, `Integer (String s)`) Ist dies nicht möglich so wird eine `NumberFormatException` ausgelöst. Dies ist der liegt vor, wenn der aus der Zeichenkette extrahierte Wert den zulässigen Gültigkeitsbereich (d.h. er liegt unter `MIN_VALUE` oder oberhalb `MAX_VALUE`) überschreitet. Ebenso wenn die Zeichenkette Symbole enthält, die nicht in eine numerische Darstellung überführt werden können.

- **compareTo**

Vergleicht den gekapselten Wert zweier Wrappertyp-Objekte.

Im Gegensatz hierzu vergleicht `equals` die Objekte direkt.

```
(1) public class WrapperComparison {
(2)     public static void main(String[] args) {
(3)         Integer i1 = new Integer ( 42 );
(4)         Integer i2 = new Integer ( 42 );
(5)
(6)         System.out.println("i1==i2="+ (i1==i2) );
(7)         System.out.println("i1.equals(i2)="+ (i1.equals(i2) ));
(8)         System.out.println("i1.compareTo(i2)="+ (i1.compareTo(i2)));
(9)     } //main()
(10)} //class WrapperComparison
```

Beispiel 44: Vergleich von Wrapperobjekten [WrapperComparison.java](#)

BildschirmAusgabe:

```
i1==i2=false
i1.equals(i2)=true
i1.compareTo(i2)=0
```

Hinweis:

Obwohl die Semantik der Operation `equals` für Objekte der Klasse `Object` und deren Subklassen als *größtmögliche unterscheidende Äquivalenzrelation* (*im Original*) festgelegt ist weicht z.B. die Implementierung in der Standard-API-Klasse `String` von dieser Maßgabe ab. Sie liefert bereits `true` wenn die beiden Zeichenketten inhaltsgleich sind, jedoch verschiedene Objekte darstellen.

- **parse...-Methode**

Die Fähigkeit der Konstruktoren, Wrapperobjekte -- nach vorheriger Gültigkeitsprüfung -- aus Zeichenketten zu erzeugen steht auch losgelöst von der Objektkonstruktion zur Verfügung.

Auf allen numerischen Wrappertypen werden `parseT`-Methoden angeboten, wobei *T* den konkreten Primitivtypen bezeichne. Diese liefern, im Falle der Gültigkeit, aus einer Zeichenkette den jeweiligen Primitivtypen zurück.

Beispiele: `parseByte (String s)`, `parseByte(String s, int radix)`, `parseInt (String s)`, `parseInt (String s, int radix)`, etc.)

- **valueOf**

Für alle Wrappertypen [integraler Primitivtypen](#) ist mit der `valueOf`-Methode eine Factorymethode zur Erzeugung eines neuen Wrapperobjektes aus einer Zeichenkette definiert.

- Veränderbare Wrapperklassen für die primitiven Typen stehen mit den *CORBA Holdern* zur Verfügung.

(siehe [BooleanHolder](#), [byteHolder](#), [ShortHolder](#), [IntHolder](#), [LongHolder](#), [FloatHolder](#), [DoubleHolder](#)).

- Die üblichen arithmetischen Operationen stehen zur Verknüpfung von Wrapperobjekten nicht zur Verfügung (kein Operator overloading!).

▲ 2.4.7 Boxing / Unboxing

Zwar realisiert Java eine grundlegende Trennung zwischen Primitivtypen und Klassen, die Wertausprägungen nicht als Objekte auffaßt. Dennoch wird diese Maßgabe durch das in der Sprachversion 1.5 eingeführte dynamische Umwandlung zwischen primitiven Werten und den zugehörigen Wrapperklassen aufgeweicht. Die Integration dieses als *Boxing/Unboxing* bezeichnete Verfahren trägt den gesammelten Anwendererfahrungen Rechnung, die eine Vereinfachung der aufwendigen Wrapper-Objekterzeugung bzw. Wertextraktion aus bestehenden Wrapperobjekten fordern. Das namensgebende Begriffspaar bezeichnet hierbei diese beiden Schritte, d.h. *Boxing* die automatische Wrapper-Objekterzeugung bzw. *Unboxing* die Wandlung eines objektgekapselften Wertes in einen Primitivwert.

Die automatische Typwandlung wird in Java ausschließlich für Übergabeparameter angeboten, wie das nachfolgende Beispiel zeigt:

```
(1) public class BUBTest1 {
(2)     public static void main(String[] args) {
(3)         BUBTest1 obj = new BUBTest1();
(4)         obj.boxIt( new Integer(42) ); //nothing new
(5)         obj.boxIt( 42 ); //dynamic boxing
(6)
(7)         obj.unBoxIt( 42 ); //nothing new
(8)         obj.unBoxIt( new Integer(42) ); //dynamic unboxing
(9)     } //main()
(10)     public void boxIt(Integer i) {
(11)         System.out.println("value="+i);
(12)     } //boxIt
(13)     public void unBoxIt(int i) {
```

```
(14)         System.out.println("value="+i);
(15)     } //unBoxIt
(16)} //class BUBTest1
```

Beispiel 45: Dynamisches Boxing/Unboxing [BUBTest1.java](#)

Das Beispiel zeigt neben den jeweils singnaturkonformen Aufrufen auch die Verwendung der dynamischen Typwandlung; für die Methode `boxIt` die automatische Erzeugung eines innerhalb des Methodenrumpfes referenzierten Wrapper-Objekts bzw. für `unBoxIt` die Extraktion des im übergebenen Wrapper-Objekt enthaltenen `int` Primitivwertes.

Der Boxingvorgang kann außer für die Wrapper-Typen auch für Objekte der Klasse `Object` durchgeführt werden, da diese als Superklasse aller Wrapper-Typen angelegt sind ist diese Konversion stets typkonform. Das nachfolgende Beispiel zeigt die Anwendung:

```
(1)public class BUBTest2 {
(2)    public static void main(String[] args) {
(3)        BUBTest2 obj = new BUBTest2();
(4)        obj.boxIt( new Integer(42) ); //nothing new
(5)        obj.boxIt( 42 ); //dynamic boxing
(6)
(7)    } //main()
(8)    public void boxIt(Object i) {
(9)        System.out.println("value="+i);
(10)    } //boxIt
(11)} //class BUBTest2
```

Beispiel 46: Dynamisches Boxing für den Typ `Object` [BUBTest2.java](#)

Das Beispiel zeigt die bisher schon mögliche Verwendung einer Ausprägung des Typs `Integer` an einer Stelle, an der eine Ausprägung von `Object` erwartet wird, was unter Nutzung der Subklassenpolymorphie möglich ist. Ausgehend von diesem Sachstand ist die Realisierung des dynamischen Boxings im Beispiel zu verstehen. Das Auftreten der `int`-Zahl wird automatisch in eine Ausprägung der Klasse `Integer` konvertiert, die dann typkonform anstelle der erwarteten `Object`-Instanz verwendet werden kann.

Die umgekehrte Nutzung, d.h. die Verwendung einer Ausprägung von `Object` an einer Stelle, an der eine konkrete `int`-Zahl erwartet wird, ist --- nach Maßgabe der nicht typsicher möglichen Konversion entgegen der Vererbungsrichtung --- nicht möglich.

▲ 2.4.8 Vererbung

Wie in objektorientierten Sprachen üblich, unterstützt auch Java das Konzept der Vererbung. Die wesentlichen Hintergründe und Anwendungsgebiete, sowie Designaspekte sind aus früheren Lehrveranstaltungen bekannt, und werden daher an dieser Stelle nicht mehr wiederholt.

Charakteristika:

- Anders als C++ unterstützt Java **nur Einfachvererbung!**
- Syntaktisch wird Vererbung durch das Schlüsselwort `extends` ausgedrückt.
- Zur **Schnittstellenvererbung** (in C++ üblicherweise als *pure virtual class* realisiert) existiert mit den [Interfaces](#) ein explizites Sprachkonzept
- Subklasse erbt nicht private Methoden und Attribute der Superklasse.
- Vererbung wird implizit als **Spezialisierung** aufgefaßt.
Jede Ausprägung einer spezialisierten Klasse kann daher an jeder Stelle im Programm verwendet werden, an der eine Ausprägung der Superklasse erwartet wird (Liskov'sches Substitutionsprinzip). Hierbei erfolgt eine implizite Typumwandlung (*up cast*).
Die originale Typinformation bleibt dessen ungeachtet erhalten.
- Umwandlung einer Subklassenausprägung in eine der Superklasse bedarf eines expliziten Cast-Statements.
Schlägt dieses fehl, wird eine [ClassCastException](#) ausgelöst (C++ retourniert hier ein null Objekt).

```
(1)public class InheritanceDemo {
(2)    public static void main(String[] args) {
(3)        C2 myC2 = new C2();
(4)
(5)        System.out.println("attrib2="+myC2.attrib2);
(6)        System.out.println("inherited attrib1="+myC2.attrib1);
(7)
(8)        C1 myC1 = myC2; //implicit type cast (up cast!)
(9)
(10)       C1 myC11 = new C1();
(11)
(12)       C2 myC22 = (C2) myC11; //explicit type cast needed (down cast!)
(13)                                           //will throw a
ClassCastException
(14)
```

```

(15)    } //main()
(16)} //class InheritanceDemo
(17)
(18)class C1 {
(19)    public int attrib1=1;
(20)} //class C1
(21)
(22)class C2 extends C1 {
(23)    public int attrib2=2;
(24)} //class C2

```

Beispiel 47: Erzeugung einer ClassCastException [InheritanceDemo.java](#)

Die Umwandlung des C2 Objektes in eines vom Typ C1 erfolgt implizit und automatisch.

Beim Versuch ein Objekt der Klasse c1 in ein Objekt der Klasse C2 umzuwandeln (*down cast* -- Subklassenobjekt wird in Superklassenobjekt gewandelt) wird eine ClassCastException erzeugt.

- Dynamische Bindung von nicht statischen Methoden (analog virtueller Funktionen in C++). Dieses Verhalten kann durch Kennzeichnung der Methode als [final](#) unterbunden werden. Klassenmethoden sind ebenso von diesem Verhalten ausgenommen.

```

(1)public class DynamicBinding {
(2)    public static void main(String[] args) {
(3)        C1 myC1 = new C1();
(4)        myC1.hello();
(5)
(6)        myC1 = new C2();
(7)        myC1.hello();
(8)
(9)        System.out.println("invoking sHello...");
(10)       ((C2) myC1).sHello();
(11)    } //main()
(12)} //class DynamicBinding
(13)
(14)class C1 {
(15)    public void hello() {
(16)        System.out.println("Hello from class C1");
(17)    } //hello()
(18)} //class C1
(19)
(20)class C2 extends C1 {
(21)    public void hello() {
(22)        System.out.println("Hello from class C2");
(23)    } //hello()
(24)
(25)    public void sHello() {
(26)        super.hello();
(27)        hello();
(28)    } //superHello()
(29)} //class C2

```

Beispiel 48: Dynamische Bindung von Methoden [DynamicBinding.java](#)

BildschirmAusgabe:

```

$java DynamicBinding
Hello from class C1
Hello from class C2
invoking sHello...
Hello from class C1
Hello from class C2

```

(Relevanter Teil der Ausgabe: oberhalb von *invoking sHello*) Trotz der Typisierung der Variable myC1 als C1-Ausprägung wird die Methode hello von C2 ausgeführt, wenn der Inhalt der Variable auf ein solches Objekt verweist.

- Das Schlüsselwort `super` dient als Platzhalter zur dynamischen Referenzierung des Superklassenobjekts. Das vorhergehende Beispiel zeigt die Verwendung in der Methode `sHello`, die zunächst auf dem Superklassenobjekt die Methode `hello` ausführt, und im Anschluß die gleichnamige auf dem aktuellen Objekt.
Anmerkung: ein Zugriff auf die *Großelternklasse*, d.h. über mehrere Hierarchieebene hinweg -- etwa durch Konkatenation von `super` zu `super.super.foo()` --, ist nicht möglich.
- Innerhalb von Konstruktoren, und dort nur als erstes Statement, steht die Surrogatmethode `super(...)` zur Verfügung, durch welche der entsprechende Superklassenkonstruktor aufgerufen werden kann. Wird der Konstruktor der Superklasse nicht explizit aufgerufen, so erfolgt automatisch ein Aufruf des parameterlosen Standardkonstruktors der Superklasse.
- Destruktorenaufrufe werden (analog C++) nicht an die Superklasse propagiert.

```

(1)public class ConstDestProp {
(2)    public static void main(String[] args) {
(3)        System.out.println("first creation...");
(4)        new C2();
(5)        System.gc();
(6)
(7)        System.out.println("second creation...");
(8)        new C2(42);
(9)
(10)       System.out.println("third creation...");
(11)       new C2(3.14);
(12)    } //main()
(13)} //class ConstDestProp
(14)
(15)class C1 {
(16)    public C1(int i) {
(17)        System.out.println("constructor of C1 executed with param i="+i);
(18)    } //C2(int)
(19)
(20)    public C1() {
(21)        System.out.println("constructor of C1 executed");
(22)    } //C1()
(23)
(24)    public void finalize() {
(25)        System.out.println("destructor of C1 executed");
(26)    } //finalize()
(27)} //class C1
(28)
(29)class C2 extends C1 {
(30)    public C2() {
(31)        System.out.println("constructor of C2 executed");
(32)    } //constructor
(33)
(34)    public C2(int i) {
(35)        super(i++);
(36)        System.out.println("constructor of C2 executed with param i="+i);
(37)    } //constructor
(38)
(39)    public C2(double d) {
(40)        System.out.println("constructor of C2 executed with param d="+d);
(41)    } //constructor
(42)
(43)    public void finalize() {
(44)        System.out.println("destructor of C2 executed");
(45)    } //finalize()
(46)} //class C1

```

Beispiel 49: Propagierung von Konstruktoren- und Destruktoreaufrufen [ConstDestProp.java](#)

BildschirmAusgabe:

```

$java ConstDestProp
first creation...
constructor of c1 executed
constructor of c2 executed
destructor of c2 executed
second creation...
constructor of c1 executed with param i=42
constructor of c2 executed with param i=43
third creation...
constructor of c1 executed
constructor of c2 executed with param d=3.14

```

Zunächst wird ein Objekt der Klasse C2 über den Aufruf des parameterlosen Konstruktors erzeugt. Vor Ausführung des Konstruktors von C2 wird durch den Übersetzer ein Aufruf an den Superklassenkonstruktor von C1 generiert. Bei Entfernung des C2-Objektes aus dem Speicher wird dessen Destruktor, nicht jedoch der der Superklasse, aufgerufen.

Die zweite Objekterzeugung nutzt den expliziten parametrisierten Konstruktor C2(int). Innerhalb dieser Methode wird zunächst explizit der Konstruktor identischer Parameterliste der Superklasse explizit per super-Schlüsselwort aufgerufen.

Die dritte Sequenz nutzt den zweiten parametrisierten Konstruktor in C2. In dessen Rumpf ist jedoch kein expliziter Aufruf eines Superklassenkonstruktors plziert. Daher wird automatisch ein Aufruf an den parameterlosen Konstruktor der Superklasse erzeugt.

- Methoden identischer Signatur überschreiben das Pendant der Superklasse.
- Sichtbarkeitsbeschränkungen an Methoden und Attribute können in der Subklasse gegenüber der Superklasse erweitert werden.
Beispiel: Superklasse: protected, Subklasse: private ist möglich.
- **Abstrakte Klassen** werden durch Angabe des Schlüsselwortes abstract als nicht instanzierbar gekennzeichnet.
- Anders als in C++ besitzt jede Javaklasse eine Superklasse. Ist eine solche nicht explizit angegeben, so setzt der

Compiler, außer für die Klasse `Object` selbst, die API-Klasse [Object](#) an diese Stelle.

Anmerkung: Gewisse generische Möglichkeiten (z.B. Objektsammlungen oder [Collections](#)) werden hierdurch erst eröffnet.

- Die Vererbungsemantik gilt auch für Arrays!
So kann jeder beliebige Array `T[]` in `Object[]` umgewandelt werden (*up cast*).

▲ 2.4.10 Schnittstellen

In C++ nicht explizit präsent. Dort wird die Schnittstellensemantik durch *pure virtual classes* (abstrakte Klasse die nur abstrakte Methoden besitzt) nachgebildet.

Hinweis: Trotz der teilweise weit gefaßten Interpretation des Begriffes *Schnittstelle* wurde diese Übersetzung hier für das originalsprachliche *Interface* verwendet.

Java-Schnittstellen versammeln [Operationen](#), d.h. Methodensignaturen ohne eine Implementierung vorzugeben, sowie Konstante.

Zusätzlich können auch Konstanten definiert werden.

Syntax:

```
InterfaceDeclaration:
    InterfaceModifiersopt interface Identifier
    ExtendsInterfacesopt InterfaceBody
```

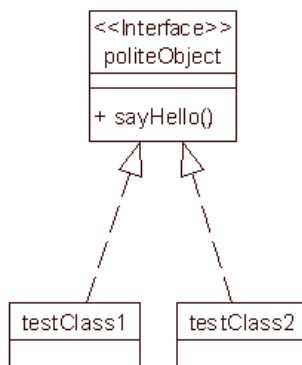
Auch für Schnittstellen stehen die bekannten Modifier `public`, `protected`, `private`, `abstract`, `static` und `strictfp` zur Verfügung.

Die Sichtbarkeitseinschränkungen sind genauso definiert wie die gleichnamigen Pendants für Klassen. Konsequenterweise muß ein `public Interface` auch in einer gleichnamigen Quelldatei untergebracht werden. Zwar definiert die Sprachspezifikation (aus Konsistenzgründen zur Definition der Klasse) den **Modifier abstract**, dieser ist aber für alle Schnittstellen implizit. Die [Sprachspezifikation](#) rät sogar explizit von seiner (verwirrenden) Verwendung ab.

Weiterführende Information: [Java Language Specification](#)

Charakteristika:

- Verhalten identisch zu abstrakten Klassen (Vererbung, Substitution, nicht instanzisierbarkeit).
- Während des Übersetzerlaufes wird geprüft, ob eine Klasse alle Operationen der Schnittstelle implementiert. Ist dies nicht der Fall, so wird die Deklaration der Klasse als abstrakt erzwungen.
- Auf Ebene der ist Mehrfachvererbung (genauer: Mehrfachimplementierung) zugelassen. Mehrfach geerbte signaturgleiche Operationen werden zu einer einzigen vereinigt.
Verursacht durch die (möglicherweise) intendierte unterschiedliche Semantik der verschiedenen implementierten Schnittstellen kann dies zu ernsthaften Problemen führen!
- Durch Schnittstellenimplementierung erhält die Klasse als zusätzlichen Typ den (Namen) der Schnittstelle.
- Attribute einer Schnittstelle sind implizit `final` `static`, und damit als Konstanten, deklariert.
- Implementiert eine Klasse eine Schnittstelle, so vererbt sich dies an alle Subklassen.
- Operationen einer Schnittstelle sind implizit als `abstract`, und damit -- innerhalb der Schnittstelle als -- nicht implementierbar, deklariert.
- Schnittstellen können andere Schnittstellen beerben. Auch hier gilt Einfachvererbung.
- Analog zu den [anonymen inneren Klassen](#) können auch anonyme innere Interfaces erstellt werden. Dasselbe gilt für benannte innere Klassen.



```

(1)public class Interface1 {
(2)    public static void main(String[] args) {
(3)        TestClass1 tco1 = new TestClass1();
(4)        TestClass2 tco2 = new TestClass2();
(5)        tco1.sayHello();
(6)        tco2.sayHello();
(7)
(8)        if (tco1 instanceof TestClass1)
(9)            System.out.println("tco1 is instance of TestClass1");
(10)
(11)        if (tco1 instanceof PoliteObject)
(12)            System.out.println("tco1 is instance of PoliteObject");
(13)
(14)        PoliteObject po;
(15)        po = tco1;
(16)        po.sayHello();
(17)        po = tco2;
(18)        po.sayHello();
(19)    } //main()
(20)} //class Interface1
(21)
(22)interface PoliteObject {
(23)    public void sayHello();
(24)} //interface PoliteObject
(25)
(26)class TestClass1 implements PoliteObject {
(27)    public void sayHello() {
(28)        System.out.println("Hello World");
(29)    } //sayHello()
(30)} //class TestClass1
(31)
(32)class TestClass2 implements PoliteObject {
(33)    public void sayHello() {
(34)        System.out.println("Guten Tag");
(35)    } //sayHello()
(36)} //class TestClass2

```

Beispiel 50: Verwendung von Schnittstellen [Interface1.java](#)

BildschirmAusgabe:

```

$java Interface1
Hello World
Guten Tag
tco1 is instance of TestClass1
tco1 is instance of politeObject
Hello World
Guten Tag

```

Das Beispiel definiert die Schnittstelle PoliteObject welche die Operation sayHello anbietet. Die beiden Klassen TestClass1 und TestClass2 implementieren jeweils Methoden für die durch die Schnittstelle definierten Operationen.

Der erste Anweisungsblock zeigt die (simple) Ausführung der genannten Methode.

Die darauffolgende Codesequenz hebt den Aspekt der Mehrfachtypisierung (Objekt von Schnittstellen-implementierender Klasse ist sowohl Ausprägung der Klasse selbst (im Beispiel: TestClass1, als auch der Schnittstelle (PoliteObject)).

Abschließend wird eine Variable vom Typ der Schnittstelle deklariert, und zunächst mit der Referenz auf ein Objekt der Klasse TestClass1 belegt. Der (statisch typischer mögliche) Aufruf sayHello ist auch hier möglich. Gleiche Bedingungen herrschen nach der Zuweisung eines Objektes vom Typ TestClass2 an dieselbe Variable.

Einige bekannte Schnittstellenverwendungen:

- [Comparable](#) -- wird u.a. von allen numerischen Wrappertypen implementiert.
- [Serializable](#) -- dient zur Speicherung (Serialisierung) von Objekten.

▲ 2.4.10 Pakete

Pakete stellen eine Sammlung von Klassen und Schnittstellen dar, die Zugriffsschutz und Namensraumverwaltung bietet.

[siehe Java Language Specification](#)

Aus dieser Definition heraus sind die Java-Pakete mit den aus C/C++ bekannten Header- und Includedateien nur schwer vergleichbar. Anders als in C/C++ werden diese externen Quellcodesequenzen nicht physisch in den zu übersetzenden Strom eingebunden, sondern existieren weiter und unabhängig. Technisch stellen Pakete ein eigenständiges Sprachmittel dar.

Insbesondere dienen sie nicht zur Separierung von Schnittstellendefinition (.h-Datei) und deren Implementierung.

Syntax einer Paketdefinition:

`package packageName;` am Anfang der Quellcodedatei.

Syntax einer Paketdefinition:

`import packageName.subPackage.subSubPackage... className;`

Hinweis: Im Gegensatz zur C/C++-Präprozessoranweisung `include` stellt die `import`-Definition einen syntaktisch korrekten Javaausdruck dar, der durch ein Semikolon abgeschlossen werden muß.

Paketbenennung: Um Pakete weltweit eindeutig identifizieren und unterscheiden zu können hat sich als Konvention ein an die URL-Notation ([IETF RFC 1738](https://www.rfc-editor.org/rfc/1738)) angelehntes Namensschema durchgesetzt. Hierbei werden die URL-Komponenten ausgehend von der Toplevel-Domain von rechts nach links angegeben.

Beispiel: Ein Paket `testPackageA` im Namensraum `myCompany.germany.org` würde als `org.germany.myCompany.testPackageA` abgelegt.

Die einzelnen Hierarchieebene der Paketidentifikation werden im JDK durch Kataloge im Dateisystem nachgebildet in denen die Javodateien des entsprechenden Paketes abgelegt werden müssen. So würde die Datei `testPackageA.class` im Katalog `/com/germany/myCompany` plazierte (Durch Substitution der Punkte im vollqualifizierten Klassennamen durch den Verzeichnistrenner `/` ergibt sich der vollqualifizierte Dateipfad.

Das JDK erfordert es, alle Quellcodedateien einer Paketebene gemeinsam zu übersetzen. Compileraufruf `javac File1.java File2.java File3.java ...`

[Weiterführende Information](#)

Charakteristika:

- Eine Javodatei kann maximal ein Paket beinhalten.
- Jede Quellcodedatei kann beliebig viele Pakete importieren.
- Pakete können geschachtelt werden; d.h. ein Paket kann neben Klassen und Schnittstellen auch weitere Pakete beinhalten.
- Klassen und Schnittstellen in Quellcodedateien ohne explizite einleitende Paketdefinition werden einem unbenannten Standardpaket (*default package*) zugeordnet.
- Alternativ zur Ablage der Paketstruktur im Dateisystem kann der entsprechende Dateibaumast auch zu einer einzigen JAR-Datei (intern ZIP-Komprimierung) zusammengefaßt werden.
- Import kann verschieden granular angegeben werden:
 - Vollqualifiziert bis auf Klassenebene -- importiert wird (ausschließlich) die spezifizierte Klasse. Der Importvorgang erfolgt erst beim tatsächlichen Zugriff (*import on demand*)
 - Auf beliebiger Ebene endend, mit Stern `*` terminiert -- importiert werden alle Unterpakete einschließlich der dort definierten Klassen.
- Für Pakete gelten besondere [Sichtbarkeitseinschränkungen](#). Besonders hervorzuheben ist die Zugriffsbehandlung nicht `public` deklarierter Attribute oder Methoden innerhalb eines Paketes. Hier werden alle Elemente desselben Paketes als vertrauenswürdig (engl. *trusted*) angesehen, und erhalten daher auch Zugriff auf als `protected` deklarierte Information.
- Auf Klassen und Schnittstellen importierter Pakete kann ohne Qualifizierung zugegriffen werden.
- Zum Übersetzungszeitpunkt wird die Namenseindeutigkeit der importierten Elemente geprüft. Liegen mehrdeutige Namensdeklarationen vor, so wird dies durch den Compiler gemeldet.

Hinweise:

- Das Paket [java.lang](#) wird durch den Übersetzer automatisch importiert. Daher stehen Methoden wie [System.out.println](#), [System.gc](#) ohne vorhergehende `import`-Anweisung zur Verfügung.
- Inhalte des Java Extension Paketes `javax` sind nicht Bestandteil des offiziellen JDKs und stehen daher oftmals nicht auf allen unterstützten Plattformen zur Verfügung.
- Die Umgebungsvariable `CLASSPATH` muß zwingend zumindest Dateisystemkataloge der benutzerdefinierten Pakete umfassen. In älteren JDK-Versionen (< v1.3) zusätzlich noch die Systempakete, bzw. die Datei `classes.zip`.

Beispiele aus der Java-API:

- [Übersicht der Pakete](#)
- [java.lang](#), enthält u.a. die [Wrappertypen](#)
- [java.applet](#), Grundpaket der [Applet-Programmierung](#)
- [java.io](#), [Ein-/Ausgabebehandlung](#)
- [java.util](#), u.a. [Collection-API](#)

Beispiel:

Quellcodedateien:

```
(1)package testPackage;
(2)
(3)public class TestClass {
(4)    public static void sayHello() {
(5)        System.out.println("hello from TestClass contained in testPackage");
(6)    } //sayHello()
(7)} //class TestClass
```

Beispiel 51: Klasse `TestClass` innerhalb des Paketes `testPackage` [TestClass.java](#)

```

(1)import testPackage.*;
(2)
(3)public class PackageUser {
(4)    public static void main(String[] args) {
(5)        testClass.sayHello();
(6)    } //main()
(7)} //class PackageUser

```

Beispiel 52: Paketverwendende Datei PackageUser.java [PackageUser.java](#)

Dateiplazierung:

TestClass.java im Verzeichnis testPackage.

packageUser.java im logisch direkt übergeordneten Verzeichnis.

Anmerkung:

Der allgemeine Import aller in testPackage enthaltenen Klassen, Schnittstellen und Pakete könnte auch per `import testPackage.*` auf die gewünschte Klasse TestClass eingeschränkt werden.

Ebenso würde der vollqualifizierte Methodenaufruf mit `testPackage.TestClass.sayHello()`; ohne `import`-Deklaration auskommen.

Statische Imports

Mit der Überarbeitung zur Sprachversion 1.5 wurde durch die *statischen Importe* eine abkürzende Schreibweise für importierte statische Methoden etabliert.

Durch die zusätzliche Angabe des Schlüsselwortes `static` vor der vollqualifizierten Klassenidentifikation stehen als Resultat alle als statisch deklarierten Methoden zum direkten Aufruf, d.h. ohne den Zwang die deklarierende Klasse zu explizieren, zur Verfügung.

Semantisch entspricht die neu geschaffene Importvariante jedoch der bisher vorstellten Mimik. Sie ergänzt diese lediglich um eine Schreibweise die syntaktisch an den Aufruf globaler Funktionen in C/C++ erinnert.

Das nachfolgende Beispiel zeigt die Anwendung beim Aufruf der Methode [sin](#) der Standardklasse [Math](#):

```

(1)import static java.lang.Math.*;
(2)
(3)public class SITest {
(4)    public static void main(String[] atrgs) {
(5)        System.out.println("sin(42)="+ sin(42) );
(6)    } //main()
(7)} //class SITest

```

Beispiel 53: Statischer Import [SITest.java](#)

▲ 3 Die Java-Plattform

▲ 3.1 Die Laufzeitumgebung

▲ 3.1.1 Garbage Collection

Wie bereits im [Einführungskapitel](#) angeschnitten, verfügt die JavaLaufzeitumgebung über eine Speicherverwaltung automatischer Speicherbereinigung (engl. *garbage collection*) des dynamisch verwalteten Heaps.

Generell kann Speicher durch den Programmierer nur konsumiert, nicht jedoch wieder freigegeben werden. (*Zur Erinnerung:* Die [explizite Nullzuweisung](#) an eine Objektvariable markiert den dadurch referenzierten Speicherplatz als nicht mehr benötigt, gibt ihn jedoch nicht direkt frei.)

Speicherbereiche werden in Java durch das Java-Schlüsselwort `new`, für Java-Objekte, bzw. durch die [explizite Definition von primitiven Datentypen](#), reserviert. Konkret geschieht die Speicherreservierung auf dem Heap ausschließlich durch die Byte-Code-Instruktionen `new`, `newarray`, `anewarray` und `multianewarray`.

Explizite Freigabemöglichkeiten, wie durch `delete` in C++, existieren nicht.

Zum Begriff *garbage collection*: Die Wortwahl impliziert, daß nicht mehr benötigter Speicher als Abfall behandelt wird, der wegzuerwerfen ist. Jedoch implementiert die automatische Speicherbereinigung nicht das intuitiv damit verknüpfte Bild des weggebens...

Vielmehr führt die *garbage collection* ein *Speicher Recycling* durch, in dessen Verlauf wiederverwendbare Speicherbereiche erkannt, und einer neuen Nutzung zugeführt werden.

Unabhängig von der tatsächlichen Implementierung vollzieht sich die Speicherbereinigung in zwei Schritten:

- Identifikation von nicht mehr erreichbaren Speicherobjekten.
- Freigabe der ermittelten Speicherobjekte.

Als zusätzliche Implementierungsrestriktion tritt unter praktischen Gesichtspunkten noch hinzu, daß der Garbage-Collector-Lauf möglichst wenig (im Idealfalle: keinen) zusätzlichen Speicher beansprucht. Nur so kann das funktionieren auch bei knappem Speicher noch gewährleistet werden.

Für die Aufgabenstellung der automatischen Freispeichergewinnung werden daher sog. *mark and sweep*-Algorithmen eingesetzt. Die zugrundeliegende Vorgehensweise kann wie folgt beschrieben werden: während seiner periodischen Durchläufe markiert der Speicherbereinigungsprozeß alle erreichbaren Speicherobjekte (*mark*-Phase). Nach Analyse aller möglichen Zugriffspfade werden die unmarkierten (da unreferenzierten) Speicherobjekte automatisch freigegeben (*sweep*-Phase).

Während der Mark-Phase werden die Speicherreferenzen verändert, daher wird die Programmausführung zu diesem Zeitpunkt unterbrochen. Als Konsequenz hat die Speicherbereinigung meßbare Auswirkung auf die Ausführungszeit. Die Hauptzeit wird jedoch während der Sweep-Phase verbracht, deren Dauer letztlich von der Größe des zur Verfügung stehenden Arbeitsspeichers abhängt.

Die durch das Mark-and-Sweep-Verfahren bedingte zeitweilige Unterbrechung der Programmausführung ist für interaktive- und Realzeitanwendungen denkbar schlecht geeignet. Daher kommen in der Praxis, so auch in Java, zumeist modifizierte -- aber aufwendigere -- Verfahren zum Einsatz, welche einzelne Speicherobjekte dynamisch sperren. Die Modifikationen setzen an der Erkenntnis an, daß sowohl die Markierungs- als auch die Löschphase inkrementell organisiert sind, d.h. sie betreffen nicht die Gesamtheit der speicherresidenten Objekte. Daher eignen sich beide zu einer kontrolliert parallelen Ausführung, die „lediglich“ dafür Sorge tragen muß, daß die aktuell durch den Garbage Collector im Zugriff befindlichen Speicherstrukturen entsprechend gesperrt sind.

Dieses Verfahren ist der naiven Referenzzählung (separate Tabelle enthält Markierung für jedes Objekt, ob Referenzen darauf existieren) deutlich überlegen, da auch zirkulär verkettete nicht erreichbare Speicherstrukturen als unerreichbar erkannt werden können.

Das nachfolgende Beispiel legt sukzessive verschiedene Speicherstrukturen an, im Folgenden wird der Ablauf des *mark and sweep*-Algorithmus verdeutlicht:

Anmerkung: Im Vorgriff auf die Behandlung der [Collection API](#) verwendet die Implementierung der Klasse `Node` die Klasse [HashSet](#) und die Schnittstelle [Iterator](#), auf die in [Kapitel 3.2.4](#) näher eingegangen wird.

```
(1)import java.util.HashSet;
(2)import java.util.Iterator;
(3)
(4)public class GCTest4 {
(5)    static int m1;
(6)
(7)    void aMethod(Node paramNode) {
(8)        Node m2 = new Node("n1");
(9)        Node tmpNode;
(10)
(11)        tmpNode = m2.createChild("n11");
(12)        tmpNode.createChild("n111");
(13)        tmpNode = m2.createChild("n12");
(14)        tmpNode.createChild("n121");
(15)        tmpNode.createChild("n122");
(16)        tmpNode.createChild("n123");
(17)
(18)        Node m3 = new Node("n2");
(19)        tmpNode = m3.createChild("n21");
(20)        tmpNode.appendChild( m3 );
(21)
(22)        System.out.println("output just for testing reasons...");
(23)        System.out.println("name of parameter Node: "+paramNode.getName() );
(24)        System.out.println("name of Node referenced by m2: "+m2.getName() );
(25)        System.out.println("child's names:");
(26)        m2.getChildsNames();
(27)
(28)        System.out.println("name of Node referenced by m3: "+m3.getName() );
(29)        System.out.println("is n21 child of n2? "+m3.isChild("n21") );
(30)        System.out.println("is n2 child of n21? "+tmpNode.isChild("n2") );
(31)
(32)        tmpNode = new Node("n31");
(33)
(34)        ((tmpNode.createChild("n32")).createChild("n33")).appendChild(tmpNode);
(35)
(36)        tmpNode = null;
(37)        System.gc();
(38)    } //aMethod()
(39)
(40)    public static void main(String[] args) {
(41)        (new GCTest4()).aMethod(new Node("n4"));
(42)    } //end main()
(43)} //class GCTest4
```

```

(44)
(45)class Node {
(46)    HashSet childs = new HashSet();
(47)    String name;
(48)
(49)    public Node(String name) {
(50)        this.name = name;
(51)        System.out.println("created Node object named "+this.name);
(52)    } //Node(String)
(53)
(54)    public void appendChild(Node child) {
(55)        childs.add(child);
(56)    } //appendChild(Node)
(57)
(58)    public Node createChild(String name) {
(59)        Node newChild = new Node( name );
(60)        childs.add(newChild);
(61)        return newChild;
(62)    } //createChild(String)
(63)
(64)    public String getName() {
(65)        return name;
(66)    } //getName()
(67)
(68)    public boolean isChild(String name) {
(69)        Iterator childIt = childs.iterator();
(70)        while (childIt.hasNext()) {
(71)            if ( ((Node) childIt.next()).getName() == name)
(72)                return true;
(73)        } //while
(74)        return false;
(75)    } //isChild(String)
(76)
(77)    public void getChildNames() {
(78)        Node child;
(79)        Iterator childIt = childs.iterator();
(80)        while (childIt.hasNext()) {
(81)            child = (Node) childIt.next(); //explicit (down) type cast needed since
HashMap contains only Object instances
(82)            System.out.println( child.getName() );
(83)            child.getChildNames();
(84)        } //while
(85)    } //getChildNames()
(86)    protected void finalize() {
(87)        System.out.println("Node object named "+name+" freed!");
(88)    } //finalize()
(89)} //class Node

```

Beispiel 54: verschiedene Speicherstrukturen [GCTest4.java](#)

Bildschirmausgabe:

```

$java GCTest4
created node object named n4
created node object named n1
created node object named n11
created node object named n111
created node object named n12
created node object named n121
created node object named n122
created node object named n123
created node object named n2
created node object named n21
output just for testing reasons...
name of parameter node: n4
name of node referenced by m2: n1
child's names:
n11
n111
n12
n121
n122
n123
name of node referenced by m3: n2
is n21 child of n2? true
is n2 child of n21? true
created node object named n31
created node object named n32
created node object named n33
node object named n31 freed!

```

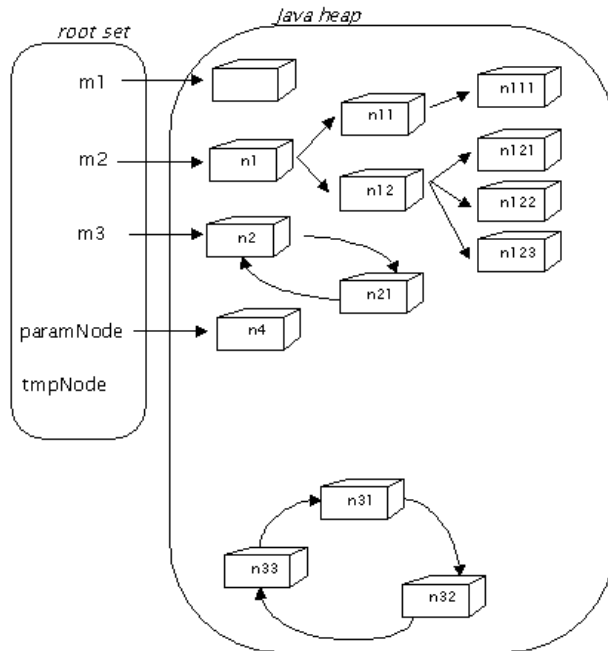
node object named n32 freed!
node object named n33 freed!

Ablauf des **mark and sweep**-Verfahrens:

Zunächst werden die Referenzen aller im Gültigkeitsbereich der aktuell ausgeführten Methode verfolgt. Im Einzelnen sind dies:

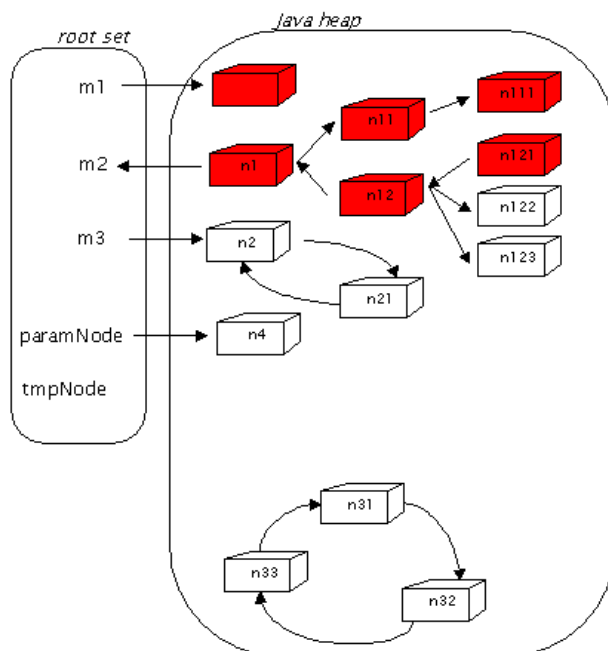
- Übergabe-Argumente
- lokale Variablen
- statische Variablen (der geladenen Klassen)
- Referenzen die durch das [Java Native Interface \(JNI\)](#) registriert sind

Die Menge dieser Speicherreferenzen wird als **root set** bezeichnet.



Mark-Phase:

Der Speicherbereinigungsprozeß durchläuft ausgehend von den Elementen des **root sets**, mit dem Ziel die erreichbaren zu markieren. Die sich zunächst intuitiv aufdrängende Vorgehensweise der rekursiven Baumtraversierung verbietet sich, wenn die **Restriktion** daß der Speicherbereinigungsprozeß nur minimale eigene Speicheranforderungen stellt berücksichtigt werden soll. (Darüberhinaus ergibt sich noch ein weit schwerwiegenderes Problem bei zyklischen Strukturen, wie wir in der Folge noch sehen werden...). Daher wird die verzeigte Speicherstruktur iterativ, unter Abhänderung der Zeigerstruktur, durchlaufen. Da der Rekursionsstack als **Gedächtnis** des genommenen Weges durch den Baum nicht zur Verfügung steht, werden die **Vorwärts**-Zeiger sukzessive zu **Rückkehr**-Zeigern modifiziert. Dies vollzieht sich in zwei Schritten. Zunächst wird der referenzierte Knoten ermittelt und zwischengespeichert. Dann wird dieser besucht und markiert; und die Zeigerrichtung invertiert. Ist ein Blattknoten erreicht und markiert, so werden die veränderten Zeigerstrukturen zur Rückkehr benutzt (sie zeigen jetzt auf den jeweils hierarchisch übergeordneten Knoten). Während des Aufsteigens werden die Zeigerstrukturen nochmals invertiert, d.h. wieder in ihren ursprünglichen Zustand (zurück) versetzt.

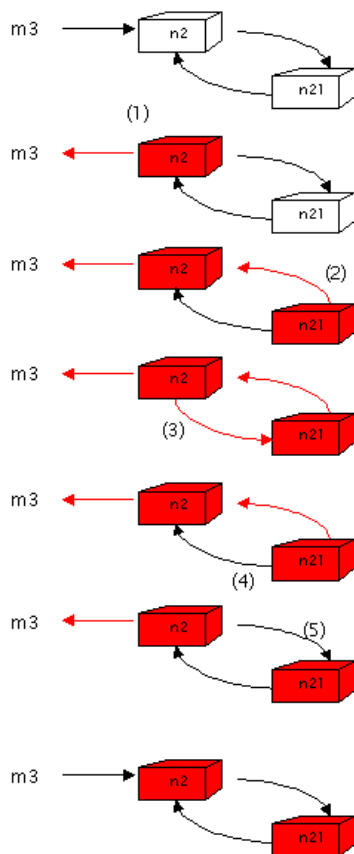


Einen Sonderfall, der bei rekursiver Implementierung aufwendig zu behandeln wäre, stellen zirkuläre Strukturen dar. Hierbei handelt es sich allgemein um Zykel beliebiger Länge. Das bedeutet, nach einer gewissen Anzahl von

Speicherknoten existiert ein Verweis (zurück) auf einen bereits traversierten Knoten. Als praktisches Beispiel solcher Strukturen seien zyklisch verkettete Listen angeführt.

Solche Strukturen sind in zweierlei Hinsicht bemerkenswert. Zum Einen, stellen sie hinsichtlich effizienter Traversierung eine Herausforderung dar, zum Anderen, da unabhängige Zyklen (siehe n31, n32 und n33 im Beispiel) nicht erreichbare, aber gültig referenzierte Strukturen sind.

Die Graphik zeigt schrittweise die Traversierung und Markierung der einzelnen Knoten eines Zyklus der Länge 2.



1. Umkehrung der Referenzierungsrichtung zwischen der Anwendervariable m3 (Element des *root set*) und dem als n2 benannten Speicherobjekt. n2 wird markiert.
2. Weiternavigation, durch Verfolgung der Referenz von n2 zum Speicherobjekt n21. Umkehrung der Referenz und Markierung des Knotens n21.
3. Verfolgung der Referenz von n21 zurück zu n2. Neuer Knoten (n2) ist bereits markiert. (Implizit ist ein Zyklus erkannt worden!) Damit sind wir zwangsläufig am Ende eines Traversierungssastes angelangt, da auf einen markierten Knoten ausschließlich markierte folgen, sofern die Hierarchie absteigend durchlaufen wird.
4. Verfolgung der Referenz zurück zum Ursprungsknoten. Anschließend: Invertierung der Verzeigerung; stellt Ursprungszustand wieder her.
5. dto.
6. dto.

Eine Abwandlung des vorhergehenden Falles zyklischer Strukturen stellt die auf n31, n32 und n33 gebildete Struktur dar.

Obwohl auf alle drei erzeugten Objekte gültige Referenzen existieren (n31 zeigt auf n32, n32 auf n33, das wiederum auf n31 verweist) sind die Objekte nach Neuzuweisung (null-Setzung) an tmpNode allesamt nicht mehr erreichbar.

Sweep-Phase:

Sie dominiert den Speicherbereinigungsablauf zeitlich. Während die Markierungen vergleichsweise schnell und effizient -- d.h. unter Vermeidung unnötiger Besuchsschritte -- angebracht werden können, wird in der zweiten Phase der gesamte Heap durchlaufen. Dabei wird jedes Speicherobjekt betrachtet, unabhängig davon ob es markiert ist oder nicht.

Nichtmarkierte Speicherobjekte werden in den Freispeicher eingereiht.

Hinweis: Die implementierungsspezifischen Aussagen beziehen sich auf SUNs Java2 (JDK v1.3) Umsetzung.

Technisch ist der Java Garbage Collector eigenständiger niederpriorer Thread innerhalb der [virtuellen Maschine](#) ausgelegt. Situations- und plattformabhängig ist dieser Thread synchron oder asynchron realisiert. Im Falle knappen Speichers, oder expliziter Anforderung durch das Programm, läuft er synchron.

Vor der Freigabe des Speicherplatzes eines Objekts wird dessen [Destruktormethode](#) aufgerufen.

Inkrementelle Speicherplatzbereinigung (auf Klassenebene) kann für die virtuelle Maschine der Fa. SUN durch die Kommandozeilenoption `-Xincgc` erzwungen werden.

Der Garbage Collector kann nicht explizit aufgerufen werden. Jedoch erwirkt der Aufruf [System.gc\(\)](#) den Versuch zur Speicherbereinigung. Nach Rückkehr der Methode muß eine Speicherbereinigung nicht zwingend erfolgt sein.

Für Objekte die bereits zur Entfernung aus dem Speicher ausgewählt wurden, jedoch die Abarbeitung ihrer Destruktoren noch aussteht, kann der Destruktlauf mit [System.runFinalization\(\)](#) angestoßen werden.

Die Java Laufzeitumgebung von SUN java erlaubt den Eingriff in die Standardgarbagecollection über folgende Kommandozeilenschalter:

- `verbose:gc`
Schaltet die Protokollierung der Garbage Collector-Aktivitäten auf Stdout ein.
- `Xnoclassgc`
Unterbindet Speicherbereinigung für Klassen.
- `Xincgc`
Aktiviert die inkrementelle Garbage Collection. Hierdurch werden zunächst die (durch Anhalten der Ausführung) spürbaren Speicherbereinigungsläufe zugunsten einer permanenten Ausführung unterbunden. Jedoch geht die Applikationsperformanz um ca. 10% zurück.

```
(1)public class GCTest1 {
(2)    public static void main(String[] args) {
(3)        System.out.println("memory before: "+Runtime.getRuntime().freeMemory() );
(4)        Test1 t1obj = new Test1();
(5)        System.out.println("memory after: "+Runtime.getRuntime().freeMemory() );
(6)        t1obj = null;
(7)        System.gc();
(8)        System.out.println("memory after garbage collection: "+ Runtime.getRuntime().
freeMemory() );
(9)    } //main()
(10)} //class GCTest1
(11)
(12)class Test1 {
(13)    double testArray[] = new double[100];
(14)    public void finalize() {
(15)        System.out.println("object of class Test1 freed");
(16)    } //finalize()
(17)} //class Test1
```

Beispiel 55: Speicherverbrauch vor und nach Garbage Collection [GCTest1.java](#)

Bildschirmausgabe:

```
$java -verbose:gc GCTest1
memory before: 1814608
memory after: 1809824
[Full GC 216K->112K(1984K), 0.0344306 secs]
object of class test1 freed
memory after garbage collection: 1915920
```

Zunächst wird der aktuelle freie Speicher innerhalb der virtuellen Maschine mittels [freeMemory\(\)](#) abgefragt (genaugenommen liefert die Methode eine Schätzung des freien Speichers in Byte). Das erzeugte Objekt initialisiert einen [double](#)-Array mit 100 Werten. Hieraus errechnet sich ein minimaler Speicherplatzbedarf von 800 Byte für das Objekt `t1obj`. Der im Beispiel ermittelte Wert von 4520 Byte wird durch weitere Effekte wie Verwaltungsstrukturen und sonstige Laufzeitinformation verursacht.

Nach Nullzuweisung und explizitem Aufruf der Garbage Collectors mit [System.gc\(\)](#) vergrößert sich der freie Speicher wieder. Er nimmt sogar über den Startwert zu. Dies ist der Freigabe von Speicherobjekten geschuldet, die nicht durch den Anwender, sondern durch die virtuelle Maschine selbst erzeugt wurden. Einen Eindruck der, üblicherweise verdeckt, automatisch geladenen Komponenten liefert der Kommandozeilenschalter `verbose:class` der der Java-Ausführungsumgebung übergeben werden kann.

Die Realisierung des Garbage Collectors von SUN erhebt nicht den Anspruch in jedem Falle Speicherplatz-optimal vorzugehen, d.h. alle potentiell unreferenzierten Objekte zu sofort entdecken und freizugeben. Dies läßt sich in einer Modifikation des obigen Beispiels zeigen:

```
(1)public class GCTest2 {
(2)    public static void main(String[] args) {
(3)        System.out.println("memory before: "+Runtime.getRuntime().freeMemory() );
(4)        Test1 t1obj = new Test1();
(5)        t1obj = null;
(6)
(7)        for (int i=0; i<10; i++) {
(8)            System.gc();
(9)            System.out.println("memory after garbage collection: "+ Runtime.getRuntime
().freeMemory() );
(10)        } //for
(11)    } //main()
(12)} //class GCTest2
(13)
(14)class Test1 {
(15)    double testArray[] = new double[100];
(16)    public void finalize() {
(17)        System.out.println("object of class Test1 freed");
(18)    } //finalize()
(19)} //class Test1
```

Beispiel 56: Mehrmaliger Garbage Collectorlauf [GCTest2.java](#)

BildschirmAusgabe:

```

$java -verbose:gc gcTest2
memory before: 1814640
[Full GC 216K->112K(1984K), 0.0336478 secs]
object of class test1 freed
memory after garbage collection: 1915920
[Full GC 113K->112K(1984K), 0.0382289 secs]
memory after garbage collection: 1915952
[Full GC 113K->112K(1984K), 0.0306936 secs]
memory after garbage collection: 1915952
[Full GC 113K->111K(1984K), 0.0392094 secs]
memory after garbage collection: 1917008
[Full GC 112K->111K(1984K), 0.0315576 secs]
memory after garbage collection: 1917008
[Full GC 112K->111K(1984K), 0.0305033 secs]
memory after garbage collection: 1917008
[Full GC 112K->111K(1984K), 0.0313825 secs]
memory after garbage collection: 1917008
[Full GC 112K->111K(1984K), 0.0304195 secs]
memory after garbage collection: 1917008
[Full GC 112K->111K(1984K), 0.0319465 secs]
memory after garbage collection: 1917008
[Full GC 112K->111K(1984K), 0.0361018 secs]
memory after garbage collection: 1917008

```

So wird ein weiterer Speicherblock, trotz der bereits erfolgten Entfernung des Objektes aus dem Speicher (siehe Ausgabe des Destruktors), der Größe 193 Bytes erst durch den vierten Garbage Collector Lauf freigegeben.

Auswirkungen auf die Programmierung:

A priori hat das Vorhandensein eines automatischen Speicherbereinigungsmechanismus keine Auswirkungen auf die Algorithmenentwicklung oder -umsetzung. Jedoch können Maßnahmen zur expliziten Freigabe nicht mehr benötigter Speicherbereiche durch den Anwendungsprogrammierer (oftmals) unterbleiben, da das Laufzeitsystem sich um die Freigabe nicht mehr erreichbarer Speicherzellen kümmert. Dies kann beispielsweise bei der Implementierung verketteter Datenstrukturen (Listen, Bäume, etc.) hilfreich sein.

Die entstehenden Algorithmen sind jedoch dann nicht mehr adaptionsfrei auf Systeme ohne garbage collection übertragbar.

Abschlußbemerkungen:

- In Einzelfällen kann der berechtigte Wunsch entstehen, Objekte freizugeben, auf die noch Referenzen existieren; beispielsweise wenn ein Objekt vollständig aus dem Speicher entfernt werden soll, obwohl es noch innerhalb von Objektsammlungen (Listen, Arrays, etc.) referenziert wird. Hierfür wurden in Java v1.2 sog. *schwache Referenzen* geschaffen, die es erlauben ausschließlich schwach referenzierte Objekte freizugeben. Implementiert ist dieses Verhalten, das so auch in anderen Programmiersprachen, wie z.B. SmallTalk verwirklicht ist, im Paket [java.lang.ref](#).
- Die technische Umsetzung der automatischen Speicherbereinigung (asynchroner Thread, sub-optimales Freigabeverhalten) bedingt insgesamt ein nichtdeterministisches Anwendungsverhalten hinsichtlich Speicherplatzbedarf der Applikation, gepaart mit einem nichtdeterministischen Laufzeitverhalten durch nicht-planbare und -vorhersehbare Aktivität des Garbage Collectors. Dies läßt die Verwendung von Java, in der Standardversion, für harte Realzeitanwendungen nicht zu. Daher haben sich in der Zwischenzeit einige Aktivitäten zur Modifikation von Java für ebendiese Anwendungsdomänen formiert.

3.1.2 Virtuelle Maschine

Kern der oft apostrophierten [Plattformunabhängigkeit](#) der Programmiersprache Java ist die Generierung eines generischen Zwischenformates -- des Byte-Codes. Dieser wird von einer plattformabhängig implementierten Programmseinheit, der *Java Virtual Machine* (Abk. JVM) interpretativ zur Ausführung gebracht. Jede Java-Applikation wird auf einer eigenen virtuellen Maschine zur Ausführung gebracht. Dies garantiert eine größtmögliche Abschottung, mit dem Ziel maximierter Sicherheit, der möglicherweise gleichzeitig auf einer realen Maschine ausgeführten Java-Programme voneinander.

Das Konzept der virtuellen Maschine, die als Programm auf einer realen Hardware abläuft, ermöglicht eine vergleichsweise einfache und schnelle Portierbarkeit auf neue Zielumgebungen, da lediglich die virtuelle Maschine an die veränderte reale Maschine angepaßt werden muß.

Der Gedanke virtueller Maschinen, die generischen Zwischencode -- oftmals auch als *P-Code* bezeichnet -- ausführen, ist nicht neu. Bereits USCD Pascal, E-BASIC und die verschiedenen SmallTalk-Implementierungen, setzt diesen praktisch um.

Die Realisierung der Befehlsfolgen (Opcodes) innerhalb der virtuellen Maschine von Java ähnelt teilweise frappant der Architektur der für die Züricher Pascal-Implementierung entwickelten (abstrakten) P-Maschine.

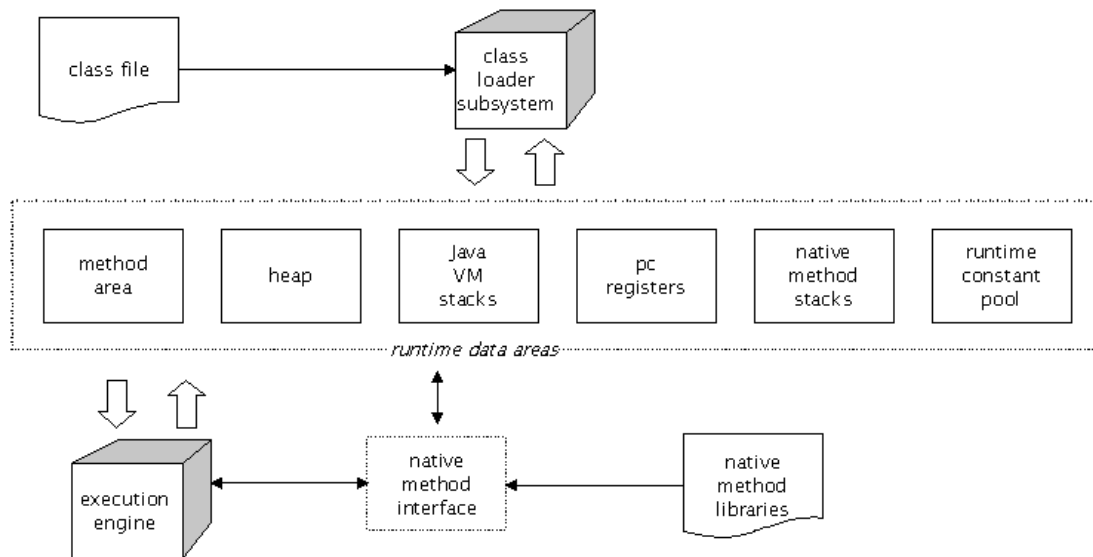
Inzwischen steht mit der [zAAP](#) (zSeries Application Assist Processor)-Hardware für die IBM-Mainframemaschinen z890 und z990 sogar eine vollständige Hardwareimplementierung der JVM zur Verfügung, welche den Charakter der *virtuellen* zu einer *realen* Maschine weiterentwickelt.

Ein Beispiel einer vollständig als Software realisierten „klassischen“ virtuellen Maschine ist [java](#), die Bestandteil des Java-Development Toolkits von SUN ist.

Bekannte andere virtuelle Maschinen sind: [Kaffe](#) oder auch [IBMs Jikes-Implementierung](#)

Wie bereits bekannt wird eine Java-Applikation durch den Aufruf `java`, gefolgt vom Namen der Startklasse und etwaiger Kommandozeilenparameter ausgeführt. Technisch gesehen bewirkt der Aufruf zunächst die Erzeugung einer neuen Instanz der virtuellen Maschine, auf welcher die Programm-Abarbeitung mit der [main-Methode](#) der Startklasse begonnen wird.

Eine Instanz einer virtuellen Maschine existiert, solange Programmfäden (engl. *Threads*) (genaugenommen: non-daemon Threads) ausgeführt werden, bzw. die virtuelle Maschine explizit beendet wird (mit dem API-Aufruf [System.exit\(\)](#)) oder ein Fehler auftritt.



Die wesentlichen Bestandteile der JVM sind:

- **method area**
Einmal vorhanden je virtueller Maschine, wird gemeinsam von allen Threads benutzt; enthält klassenspezifische Daten (Typinformation).
- **heap**
Einmal vorhanden je virtueller Maschine, wird gemeinsam von allen Threads benutzt; enthält die dynamisch erzeugten Objekte.
- **Java VM stacks**
Threadspezifisch. Enthält Zustand von nicht-nativen Methodenaufrufen, deren lokale Variablen, Übergabeparameter, den möglichen Rückgabewert sowie Methoden-interne Berechnungsergebnisse. ([vgl. JVM-Spezifikation](#))
- **pc registers**
Threadspezifisch; enthält für jeden Zeitpunkt der Ausführung einer nicht-nativen Methode die Adresse der derzeit ausgeführten Instruktion. ([vgl. JVM-Spezifikation](#)). Während der Abarbeitung nativer Methoden ist der Wert auf undefined gesetzt. Konkrete Größe des virtuellen pc-Registers hängt von der Adresslänge der realen Plattform ab.
- **native method stack**
Implementierungsspezifischer Speicherbereich zur Behandlung von nativen Methodenaufrufen.
- **runtime constant pool**
Klassen- oder Schnittstellenspezifisch. Enthält verschiedene Konstanten, die zur Übersetzungszeit feststehen und in der `constant_pool` Tabelle der class-Datei abgelegt sind. Die Funktion dieses Bereichs ähnelt dem einer konventionellen Symboltabelle. ([vgl. JVM-Spezifikation](#))

Die Java-Stacks sind in *stack frames* organisiert. Jedem Methodenaufruf ist ein [Stack-Frame](#) zugeordnet, der beim Aufruf erzeugt (push), und beim Verlassen (pop) vom Stack entnommen wird.

Innerhalb eines Frames befindet sich

- Array der lokalen Variablen
- Operanden-Stack
Wichtigste Datenstruktur innerhalb der virtuellen Maschine. Der Operanden-Stack enthält alle Eingangs- und Ausgangsdaten der verschiedenen Berechnungen. Seine Einträge sind typisiert in 32-Bit Worten organisiert; die korrekte Typisierung der Eingangsoperanden wird von jedem Opcode geprüft.
- Referenz auf runtime constant pool
- implementierungsspezifische- und debugging-Information

Den für den Anwendungsentwickler offensichtlichsten Bestandteil der virtuellen Maschine, bilden jedoch die JVM-Instruktionen -- die Maschinensprache der JVM.

Der [Befehlssatz der JVM](#) umfaßt ausschließlich genau ein Byte lange Opcodes.

Die JVM ist generell *stack-orientiert*. Dies bedeutet, daß Quell- und Zieloperanden der meisten Operationen werden vom Stack entnommen, und das Ergebnis dort abgelegt. Insbesondere existieren, abgesehen von vier Verwaltungspeicherplätzen je Ausführungs-Thread, keine virtuellen Prozessorregister, um die Implementierungsanforderungen an die reale Plattform zu minimieren.

Als threadlokale Register stehen zur Verfügung:

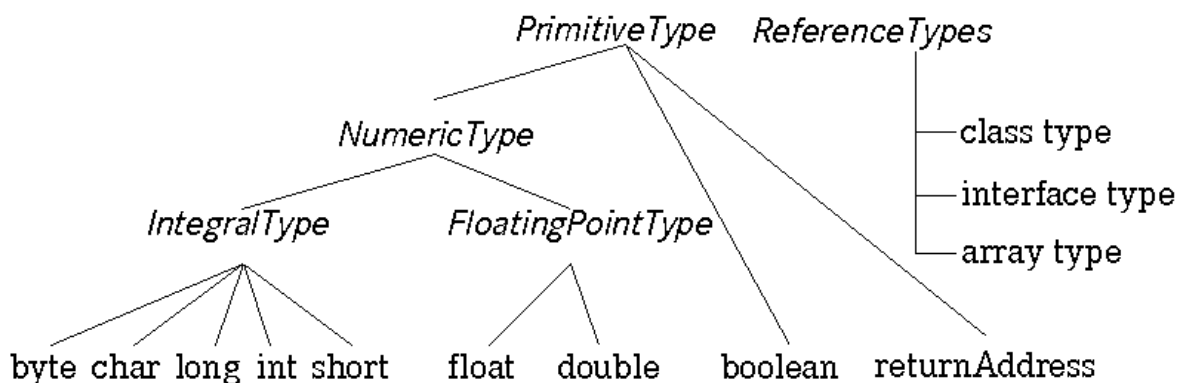
- [pc -- program counter](#)
Enthält die Adresse der gerade ausgeführten Instruktion.
Handelt es sich um eine native-Methode, die in einer anderen Programmiersprache ausgeführt wird, so ist der Wert *undefined*.

- `optop`
Verweist auf die Spitze des [Operanden Stacks](#).
- `frame`
Verweist auf die Ausführungsumgebung der derzeit aktiven Methode.
- `vars`
Ein Zeiger auf die erste lokale Variable der aktuellen Methode.

Die Adresslänge innerhalb der JVM ist auf vier Byte (32 Bit) fixiert. Hieraus ergibt sich ein (theoretischer) Adressraum von 4 GB.

Die initiale und maximale Ausdehnung des Heaps kann durch die Kommandozeilenschalter `Xms` bzw. `Xmx` gesteuert werden (Beispiel: `java -Xms350M -Xmx500M HelloWorld` führt ein einfaches [Hello-World-Beispiel](#) mit einer anfänglichen Speicherausstattung von 350 MB aus, die im Verlaufe des Programmablaufs auf höchstens 500 MB anwachsen kann.)

Das **Typsystem der JVM** lehnt sich eng an das der Hochsprache an. ([Zur Erinnerung: primitive Typen in Java](#)) Zusätzlich erweitert es die Primitivtypen um einen Adresstypen `returnAddress` und führt explizite Referenztypen auf die verschiedenen high-level Typen (Klassen, Schnittstellen, Arrays) ein.



Datentypen der JVM:

- `byte`
Vorzeichenbehaftete 8-Bit Zahl in Zweierkomplementdarstellung.
- `short`
Vorzeichenbehaftete 16-Bit Zahl in Zweierkomplementdarstellung.
- `int`
Vorzeichenbehaftete 32-Bit Zahl in Zweierkomplementdarstellung.
- `long`
Vorzeichenbehaftete 64-Bit Zahl in Zweierkomplementdarstellung.
- `char`
Vorzeichenbehaftete 16-Bit Zahl zur Darstellung von Unicode-Zeichen.
- `float`
32-Bit Binärdarstellung einer Fließkommazahl gemäß [IEEE 754-1985](#).
- `double`
64-Bit Binärdarstellung einer Fließkommazahl gemäß [IEEE 754-1985](#).
- `boolean`
Wird zwar durch die JVM definiert, jedoch existieren keine Opcodes, die Parameter dieses Typs erwarten. Der Compiler setzt Anweisungen die `boolean`-Typen enthalten als Operationen auf `int`-Typen um.
- `returnAddress`
Für den Programmierer nicht zugänglicher Typ, der Sprungadressen aufnimmt.
- Referenztypen
Verweisen auf Klassen, Arrays oder Schnittstellen.
Der Wert kann auch `null` sein, wobei die JVM keine konkrete Darstellung dieses Wertes unterstellt.

Jede Bytecode-Instruktion besteht zunächst aus ihrem Opcode, optional gefolgt von den benötigten Operanden. Diese stehen jedoch nicht für sich, sondern sind eingebettet in den organisatorischen Rahmen der `class`-Datei, deren [Format](#) im Anschluß vorgestellt wird.

Die verschiedenen Maschineninstruktionen lassen sich in Klassen einteilen:

Instruktionen zum Zugriff auf lokale Variablen:

Instruktion	Funktion
ret	Rücksprung aus Subroutine, wird in der Implementierung von finally benutzt
aload	Lädt Referenz von spezifisch indizierter Position auf den Operanden-Stack
aload <n>	Lädt Referenz auf Operanden-Stack (Index im Opcode explizit angegeben)
astore	Speichert auf Operanden Stack liegenden Wert in lokale Variable
astore <n>	Legt Referenz in lokaler Variable auf Operanden-Stack ab (Index wird im Opcode explizit angegeben)
dload	Lädt <code>double</code> -Wert aus lokaler Variable auf den Operanden-Stack

<u>dstore</u>	Lädt double-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab
<u>fload</u>	Lädt float-Wert aus lokaler Variable auf den Operanden-Stack
<u>fstore</u>	Lädt float-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab
<u>iload</u>	Lädt int auf Operanden-Stack (Index im Opcode explizit angegeben)
<u>iload <n></u>	Lädt int-Wert aus lokaler Variable auf den Operanden-Stack
<u>istore</u>	Legt int-Wert von spezifisch indizierter Position in lokaler Variable auf Operanden-Stack ab
<u>istore <n></u>	Lädt int-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab
<u>lload</u>	Lädt long-Wert aus lokaler Variable auf den Operanden-Stack
<u>lstore</u>	Lädt long-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab
<u>inc</u>	Inkrementiert lokale Variable um fixe int-Zahl

Instruktionen zur expliziten Modifikation des Operanden-Stacks:

Instruktion Funktion

<u>bipush</u>	Ablegen eines <u>byte-Wertes</u> auf dem Operanden-Stack
<u>sipush</u>	Ablegen eines <u>short-Wertes</u> auf dem Operanden-Stack
<u>pop</u>	Entnimmt und verwirft (de facto: löscht) obersten Operanden-Stack-Eintrag.
<u>pop2</u>	Entnimmt (de facto: löscht) obersten beiden Operanden-Stack-Einträge.
<u>swap</u>	Tauscht die beiden obersten Operanden-Stack-Einträge aus.
<u>ldc</u>	Ablegen eines Elements (referenziert über 16-Bit Index) des <u>runtime constant pools</u> auf dem Operanden-Stack
<u>ldc w</u>	Ablegen eines Elements (referenziert über 32-Bit Index) des <u>runtime constant pools</u> auf dem Operanden-Stack
<u>aconst null</u>	Legt null auf dem Operanden-Stack ab.
<u>dconst <d></u>	Legt double Konstante auf dem Operanden-Stack ab. dconst_0 die Konstante 0.0, bzw. dconst_1 den Wert 1.0.
<u>fconst <f></u>	Legt float Konstante auf dem Operanden-Stack ab. fconst_0 die Konstante 0.0, bzw. fconst_1 den Wert 1.0.
<u>iconst <i></u>	Legt int-Konstante auf dem Operanden-Stack ab. Es existieren Opcodes für folgende Konstanten: iconst_m1 -- -1; iconst_0 -- 0; iconst_1 -- 1; iconst_2 -- 2; iconst_3 -- 3; iconst_4 -- 4; iconst_5 -- 5.
<u>dup</u>	Dupliziert oberstes Element des Operanden-Stacks. <u>dup x1</u> legt das neue Element als zweitunterstes auf dem Stack ab. <u>dup x2</u> legt das neue Element, abhängig vom Stack-Inhalt, als zweit- oder drittunterstes auf dem Stack ab.
<u>dup2</u>	Dupliziert die beiden obersten Elemente des Operanden-Stacks. <u>dup2 x1</u> legt die neuen Elemente als zweitunterstes und folgendes auf dem Stack ab. <u>dup2 x2</u> legt die neuen Elemente, abhängig vom Stack-Inhalt, als zweit- oder drittunterstes und folgendes auf dem Stack ab.
<u>lconst <l></u>	Legt long Konstante auf dem Operanden-Stack ab. Es existieren Opcodes für folgende Konstanten: iconst_0 -- 0; iconst_1 -- 1.

Instruktion zur Steuerung des Kontrollflusses:

Instruktion Funktion

<u>goto</u>	Bedingungsloser Sprung innerhalb derselben Methode. Der anzugebende <i>branch offset</i> ist auf 16-Bit fixiert, woraus sich ableiten läßt, daß das Opcodesegment einer Methode niemals (in der JVM-Version 1.3) die Größe von 64KByte überschreiten darf. Näheres zu den Einschränkungen der virtuellen Maschine findet sich in im <u>Abschnitt 4.10 der JVM-Spezifikation</u> , sowie in der <u>Diskussion des Class-File Formats</u> .
<u>goto w</u>	Bedingungsloser Sprung innerhalb derselben Methode (mit 32-Bit Offset)
<u>if acmp<cond></u>	Bedingter Sprung im Falle der Gültigkeit der Bedingung. Die zu vergleichenden Operanden werden als Referenzen übergeben. Als Bedingungen stehen Gleichheit (eq) und Ungleichheit (ne) zur Verfügung.
<u>if icmp<cond></u>	Bedingter Sprung im Falle der Gültigkeit der Bedingung. Die zu vergleichenden Operanden werden als int-Werte übergeben. Als Bedingungen stehen zur Verfügung: Gleichheit (eq), Ungleichheit (ne), Kleiner (lt), Kleiner oder Gleich (le), Größer (gt) und Größer oder Gleich (ge).
<u>if<cond></u>	Bedingter Sprung, nach Vergleich des obersten Elements des Operanden-Stacks mit Null Für spezifische Vergleiche stehen folgende Opcodes zur Verfügung: Gleichheit (ifeq), Ungleichheit (ifne), Kleiner (iflt), Kleiner oder Gleich (ifle), Größer (ifgt), Größer oder Gleich (ifge).
<u>ifnonnull</u>	Bedingter Sprung -- im Falle der Ungleichheit -- nach Vergleich der auf dem Operanden-Stack befindlichen Referenz mit Null

<u>ifnull</u>	Bedingter Sprung -- im Falle der Gleichheit -- nach Vergleich der auf dem Operanden-Stack befindlichen Referenz mit Null
<u>jsr</u>	Unbedingter Sprung, unter Sicherung der Rücksprungadresse auf dem Operanden-Stack, zu 16-Bit Adresse.
<u>jsr_w</u>	Unbedingter Sprung, unter Sicherung der Rücksprungadresse auf dem Operanden-Stack, zu 32-Bit Adresse.
<u>lookupswitch</u>	Zugriff auf Sprungtabelle per Schlüssel und anschließende Verzweigung. Benutzt zur Implementierung des <u>switch</u> -Konstrukts
<u>tableswitch</u>	Indexbasierter Zugriff auf Sprungtabelle und anschließende Verzweigung.

Instruktionen zur Operation auf Klassen und Objekten:

Instruktion Funktion

<u>anewarray</u>	Array-Erzeugung an definierter Stelle im <u>Laufzeit-Konstanten-Pool</u> .
<u>checkcast</u>	Typkompatibilitätsprüfung (<u>siehe Beispiel</u>)
<u>instanceof</u>	Prüft ob Objekt gegebenen Typ hat (d.h. Ausprägung der Klasse -- oder einer Subklasse -- ist; das Interface implementiert; Array-Kompatibel ist)
<u>new</u>	Erzeugt neues Objekt <i>Hinweis:</i> Die Punktnotation zur Trennung der Pakethierarchien wird hier durch Slash „/“ ersetzt.

Instruktionen zur Methodenausführung:

Instruktion	Funktion
<u>invokespecial</u>	Ruft <u>Instanzenmethode</u> auf; mit besonderer Behandlung bestimmter Umstände. <i>Hinweis:</i> Diese Instruktion wurde umbenannt, frühere JDK-Versionen benutzen <code>invokevirtual</code> .
<u>invokestatic</u>	Ruft statische <u>Klassenmethode</u> auf.
<u>invokevirtual</u>	Ruft Instanzenmethode auf.
<u>invokeinterface</u>	Ruft Schnittstellenmethode auf.
<u>areturn</u>	Retourniert Referenz auf Speicherobjekt nach Methodenausführung.
<u>dreturn</u>	Retourniert double-Wert nach Methodenausführung.
<u>freturn</u>	Retourniert float-Wert nach Methodenausführung.
<u>ireturn</u>	Retourniert int-Wert nach Methodenausführung.
<u>lreturn</u>	Retourniert long-Wert nach Methodenausführung.
<u>return</u>	Retourniert nach Methodenausführung ohne Rückgabewert (void-Methode).

Instruktionen zum Zugriff auf Attribute:

Instruktion Funktion

<u>getfield</u>	Legt Attributinhalt eines Objekts auf Operanden-Stack ab. Die Klasse des Objekts, auf das der Zugriff erfolgen soll, wird über einen 16-Bit Offset auf dem <u>runtime constant pool</u> adressiert. Auch hier wird wieder die Limitierung der virtuellen Maschine auf 2 ¹⁶ Klassen deutlich.
<u>getstatic</u>	Legt Attributinhalt eines statischen Klassenattributs auf dem Operanden-Stack ab.
<u>putfield</u>	Setzt Wert eines Attributs.
<u>putstatic</u>	Setzt Wert eines statischen Klassenattributs.

Instruktionen zur Operation auf Arrays:

Instruktion	Funktion
<u>newarray</u>	Erzeugt einen neuen Array, und definiert die Komponententypen als einen der <u>Primitivtypen</u> . Die Implementierung von SUN verwendet für den Wahrheitswert je acht Bit. Anderen Umsetzungen ist es explizit Freigestellt hier mit optimierteren Speicherstrukturen zu operieren.
<u>anewarray</u>	Erzeugt einen neuen Array von Referenztypen zur Aufnahme beliebiger Objekte.
<u>multianewarray</u>	Erzeugt einen mehrdimensionalen Array.
<u>aaload</u>	Lädt Referenz von Arrayposition.
<u>aastore</u>	Speicher Objekt an spezifischer Arrayposition.
<u>arraylength</u>	Liefert Elementanzahl (Kardinalität) eines Arrays.

<u>baload</u>	Lädt byte oder boolean aus Arrayposition.
<u>bastore</u>	Legt byte oder boolean an Arrayposition ab.
<u>saload</u>	Lädt short aus Arrayposition.
<u>sastore</u>	Legt short an Arrayposition ab.
<u>caload</u>	Lädt char aus Arrayposition.
<u>castore</u>	Legt char an Arrayposition ab.
<u>daload</u>	Lädt double aus Arrayposition.
<u>dastore</u>	Legt double an Arrayposition ab.
<u>faload</u>	Lädt float aus Arrayposition.
<u>fastore</u>	Legt float an Arrayposition ab.
<u>iaload</u>	Lädt int aus Arrayposition.
<u>iastore</u>	Legt int an Arrayposition ab.
<u>laload</u>	Lädt long aus Arrayposition.
<u>lastore</u>	Legt long an Arrayposition ab.

Instruktionen zur Typkonversion:

Instruktion	Funktion
<u>i2b</u>	Konvertiert int zu byte.
<u>i2c</u>	Konvertiert int zu char.
<u>i2d</u>	Konvertiert int zu double.
<u>i2f</u>	Konvertiert int zu float.
<u>i2l</u>	Konvertiert int zu long.
<u>i2s</u>	Konvertiert int zu short.
<u>d2f</u>	Konvertiert double zu float.
<u>l2d</u>	Konvertiert long zu double.
<u>l2f</u>	Konvertiert long zu float.
<u>l2i</u>	Konvertiert long zu int.
<u>d2f</u>	Konvertiert double zu float.
<u>f2d</u>	Konvertiert float zu double.
<u>f2i</u>	Konvertiert float zu int.
<u>f2l</u>	Konvertiert float zu long.
<u>d2f</u>	Konvertiert double zu float.
<u>d2f</u>	Konvertiert double zu float.
<u>d2i</u>	Konvertiert double zu int.
<u>d2l</u>	Konvertiert double zu long.

Instruktionen zur Durchführung arithmetischer Operationen:

Eingangssperanden werden vom Stack entnommen und das Berechnungsergebnis ebenda abgelegt.

Instruktion	Funktion
<u>dadd</u>	Addiert zwei double-Werte.
<u>dsub</u>	Subtrahiert zwei double-Werte.
<u>ddiv</u>	Dividiert zwei double-Werte.
<u>dmul</u>	Multipliziert zwei double-Werte.
<u>dneg</u>	Negiert double-Wert durch Zweierkomplementbildung.
<u>drem</u>	Divisionsrest bei Division zweier double-Werte.
<u>dcmp<op></u>	Vergleicht zwei double Werte. Ist einer der beiden Operanden NaN, so legt dcmpg1, dcmpl hingegen -1 als Ergebnis auf dem Stack ab.

<u>fadd</u>	Addiert zwei float-Werte.
<u>fsub</u>	Subtrahiert zwei float-Werte.
<u>fmul</u>	Multipliziert zwei float-Werte.
<u>fdiv</u>	Dividiert zwei float-Werte.
<u>iadd</u>	Addiert zwei int-Werte.
<u>isub</u>	Subtrahiert zwei int-Werte.
<u>imul</u>	Multipliziert zwei int-Werte.
<u>idiv</u>	Dividiert zwei int-Werte.
<u>land</u>	Boole'sche UND-Verknüpfung zweier int-Werte.
<u>lor</u>	Boole'sche ODER-Verknüpfung zweier int-Werte.
<u>ixor</u>	Exklusive Boole'sche ODER-Verknüpfung zweier int-Werte.
<u>ineg</u>	Negiert int-Wert durch Zweierkomplementbildung.
<u>irem</u>	Divisionsrest bei Division zweier int-Werte.
<u>ishl</u>	Linksshift eines int-Wertes.
<u>ishr</u>	Rechtsshift eines int-Wertes.
<u>iushr</u>	Rechtsshift eines int-Wertes unter Nulleinfügung und Vorzeichenlösung.
<u>fneg</u>	Negiert float-Wert durch Zweierkomplementbildung.
<u>frem</u>	Divisionsrest bei Division zweier float-Werte.
<u>fcmp<op></u>	Vergleicht zwei float-Werte. Ist einer der beiden Operanden NaN, so legt fcmpg1, fcmpl hingegen -1 als Ergebnis auf dem Stack ab.
<u>ladd</u>	Addiert zwei long-Werte.
<u>land</u>	Boole'sche UND-Verknüpfung zweier long-Werte.
<u>ldcmp</u>	Vergleicht zwei long-Werte.
<u>ldiv</u>	Dividiert zwei long-Werte.
<u>lmul</u>	Multipliziert zwei long-Werte.
<u>lneg</u>	Negiert long-Wert durch Zweierkomplementbildung.
<u>lrem</u>	Divisionsrest bei Division zweier long-Werte.
<u>lor</u>	Boole'sche ODER-Verknüpfung zweier long-Werte.
<u>lshl</u>	Linksshift eines long-Wertes.
<u>lsqr</u>	Rechtsshift eines long-Wertes.
<u>lsub</u>	Subtrahiert zwei long-Werte.
<u>lushr</u>	Rechtsshift eines long-Wertes unter Nulleinfügung und Vorzeichenlösung.
<u>lxor</u>	Exklusive Boole'sche ODER-Verknüpfung zweier long-Werte.

Sonstige Instruktionen:

Instruktion	Funktion
Ausnahmebehandlung	<u>athrow</u> Wirft eine Exception.
Synchronisation -- Monitoroperationen	<u>monitorenter</u> Der Zugriff auf jedes Objekt wird durch einen Monitor synchronisiert. Die Anweisung sperrt ein Objekt.
	<u>monitorexit</u> Gibt ein gesperrtes Objekt frei.
Sonstige Opcodes	<u>nop</u> Buchstäblich: <i>no operation</i> . Bewirkt nichts; keine Operatoren, keine Änderungen am Operanden-Stack.

Die beiden Opcodes mit den Ordnungsnummern 254 und 255 (0xfe und 0xff, mnemonic *impdep1* und *impdep2*) sind durch SUN als reserviert gekennzeichnet. Sie können von durch den Hersteller der virtuellen Maschine mit eigendefinierter Funktionalität implementiert werden.

Darüberhinaus ist mit dem Opcode 202 (mnemonic *breakpoint*) ein Einstiegspunkt für Debugger definiert.

Alle Opcodes sind mit einem Byte codiert. Hieraus ergibt 256 als maximaler Befehlsumfang der virtuellen Maschine. Zur Verringerung der notwendigen verschiedenen Befehle sind nicht alle Opcodes für alle [Typen der JVM](#) implementiert. Üblicherweise existieren nur Opcodes für *int*, *float*, *long* und *double* sowie die Referenzen. Für alle anderen Typen stehen Konvertierungsmöglichkeiten in die genannten zur Verfügung.

Opcode	byte	short	int	long	float	double	char	Referenz
...ipush	bipush	sipush						
...const			iconst	lconst	fconst	dconst		aconst
...load			iload	lload	fload	dload		aload
...store			istore	lstore	fstore	dstore		astore
...inc			iinc					
...aload	baload	saload	iaload	laload	faload	daload	caload	aaload
...astore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
...add			iadd	ladd	fadd	dadd		
...sub			isubb	lsub	fsub	dsub		
...mul			imul	lmul	fmul	dmul		
...div			idiv	ldiv	fdiv	ddiv		
...rem			irem	lrem	frem	drem		
...neg			ineg	lneg	fneg	dneg		
...shl			ishl	lshl				
...shr			ishr	lshr				
...ushr			iushr	lushr				
...and			iand	land				
...or			ior	lor				
...xor			ixor	lxor				
i2...	i2b	i2s		i2l	i2f	i2d		
l2...			l2i		l2f	l2d		
f2...			f2i	f2l	f2d			
...cmp				lcmp				
...cmpl				fcmpl	dcmpl			
...cmpg				fcmpg	dcmpg			
if....cmpcond			if_icmpcond					if_acmpcond
...return			ireturn	lreturn	freturn	dreturn		areturn

Treten bei der Ausführung der Opcodes Ausnahmen auf, so werden durch die virtuelle Maschine [Laufzeit-Ausnahmeereignisse](#) (engl. *runtime exception*) generiert. [Wie bekannt](#) werden Ausnahmen dieser Kategorie (üblicherweise) nicht aufgefangen und behandelt. So wird die [ClassCastException](#) im [Beispiel aus Kapitel 2](#) durch die versuchte explizite Typumwandlung ausgelöst. Der erzeugte Bytecode für diese Anweisung lautet:

```
aload_3
checkcast 2
```

aload_3 lädt die Referenz auf eine lokale Variable auf den Operanden-Stack. Die lokale Variable 3 entspricht im Beispiel myC11. checkcast testet ob die auf dem Operanden-Stack befindliche Referenz kompatibel zum übergebenen Typen (hier die 2 als Referenz auf die zweite geladene Klasse; benannt mit C2) ist. Im Falle der Nichtkompatibilität wird durch die virtuelle Maschine eine ClassCastException erzeugt.

Beispiel 1: Einfache arithmetische und Ein- / Ausgabeoperationen

```
(1).class public examples/BC1
(2).super java/lang/Object
(3)
(4).method public <init>()V
(5)    aload_0
(6)    invokevirtual java/lang/Object/<init>()V
(7)    return
(8).end method
(9)
(10).method public static main([Ljava/lang/String;)V
(11)    .limit locals 5
(12)    .limit stack 10
(13)
(14)    iconst_2
(15)    istore_0
```



```

(16)
(17)    bipush 101
(18)    istore_1
(19)
(20)    bipush 99
(21)    istore_2
(22)
(23)    ;we will need this twice
(24)    getstatic java/lang/System/out Ljava/io/PrintStream;
(25)    astore_3
(26)
(27)    iload_1
(28)    iload_2
(29)    iadd
(30)    istore_1
(31)
(32)    ;convert int to string
(33)    iload_1
(34)    invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
(35)    astore 4
(36)
(37)    ;Print a string
(38)    aload_3
(39)    aload 4
(40)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(41)
(42)    iload_1
(43)    iload_0
(44)    idiv
(45)
(46)    ;convert int to string
(47)    invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
(48)    astore 4
(49)
(50)    ;Print a string
(51)    aload_3
(52)    aload 4
(53)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(54)
(55)    ;print a fixed string
(56)    aload_3
(57)    ldc "The End"
(58)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(59)    return
(60).end method
(61)

```

[Download des Beispiels](#)

Der Java-Assemblercode des Beispiels 1 zeigt die Verwendung einiger einfacher arithmetischer und Ein-/Ausgabeoperationen.

Zunächst zeigt das Beispiel den Aufbau einer Java-Assemblerdatei, wie sie vom Übersetzer [Jasmin](#) akzeptiert und in ausführbaren Java-Bytecode umgewandelt wird.

So legt die `.class`-Deklaration zunächst fest, daß es sich um die öffentlich zugängliche (d.h. als `public` deklarierte) Klasse `BC1`, im Paket `examples` handelt.

Die darauf folgende `.super`-Definition legt die Elternklasse der betrachteten Klasse fest. Im Falle keiner explizit definierten Elternklasse ist dies vorgabegemäß die Standardklasse [class](#).

Nach den einführenden Deklarationen definiert die Quellcodedatei den statischen Initialisierer.

Die Methode `main` stellt den Beginn der aktiven Verarbeitung dar. Ihre Signaturdefinition (`[Ljava/lang/String;)V` läßt die JVM-interne Kurzschreibweise des Typsystems erkennen. So deutet die in den Klammern der Übergabewerte eingeschlossene eckige Klammern an, daß ein Array gleichtypisierter Ausprägungen übergeben wird. Diese Ausprägungen sind alle vom Standardtyp `java.lang.String` (die paket-separierenden Punkte werden JVM-intern zu Pfadseparatoren aufgelöst). Zusätzlich ist dem Klassennamen ein einleitendes `L` vorangestellt, um auszudrücken, daß es sich nicht um einen Primitivtyp, sondern um eine Sprachkomponente (das „L“ deutet hierbei auf den Begriff *language* hin) handelt.

Nach der Klammer ist der Rückgabetyt --- im Falle von `main` vorgabegemäß `void` --- angegeben. Auch er wird unter Verwendung derselben Abkürzungskonvention dargestellt.

Zu Eingangs der Methode `main` allozieren die beiden Direktiven `.limit` zunächst Speicher für die lokalen Variablen (`.limit locals`) und die Tiefe des methodenintern verwendeten Operandenstacks (`.limit stack`).

Die (aktive durch den Programmierer gesteuerte) Verarbeitung beginnt im Beispiel mit dem Anweisung `iconst_2` welche den ganzzahligen Wert 2 auf dem Operandenstack ablegt. Anschließend wird dieser Wert, mittels der Anweisung `istore_0` vom Stack entnommen und in die erste lokale Variable gespeichert.

Mit [iconst_2](#) findet ein besonderer Befehl zur Ablage einer ganzzahligen Konstante auf dem Operanden Stack Verwendung, der es gestattet bestimmte (häufig benötigte) Konstantenablagen in nur genau einem Byte auszudrücken. Durch die JVM-Spezifikation vorgesehen sind hierbei Instruktionen für die Konstanten -1, 0, 1, 2, 3, 4 oder 5. Im Ergebnis ist die Nutzung der abkürzenden Befehlsschreibweise äquivalent zum Einsatz der

Instruktion [bipush](#) unter Explizierung der abzulegenden Konstante.

Diese äquivalente Form der Belegung einer lokalen Variable zeigt der zweite Anweisungsblock, der die numerische Konstante 101, für die keine abkürzende Schreibweise angeboten wird, auf dem Operandenstack ablegt um sie der zweiten lokalen Variable (mit der Indexnummer 1) zuzuweisen.

In derselben Weise wird für die Initialisierung der dritten lokalen Variablen mit dem Wert 99 verfahren.

Anschließend wird durch `getstatic` der Dateideskriptor der Standardausgabe (d.h. desjenigen Streams mit dem Wert `System/out`) gelesen und die zurückgelieferte Adresse in der vierten lokalen Variable (Indexnummer 3) abgelegt.

Der darauffolgende Anweisungsblock zeigt die Umsetzung einer einfachen Ganzzahladdition, die zunächst die beiden zu verknüpfenden Operanden (die Inhalte der lokalen Variablen mit den Indexnummern 1 und 2) auf dem Stack ablegt und anschließend mittels der Ganzzahladdition (`iadd`) verknüpft.

Das auf dem Stack abgelegte Berechnungsergebnis wird durch `istore_1` in die zweite lokale Variablen zugewiesen.

Der nächste Anweisungsblock bereitet die Ausgabe des Berechnungsergebnisses auf der Standardausgabe vor. Hierzu platziert er zunächst den Inhalt der lokalen Variable mit der Indexnummer 1 (d.h. das Berechnungsergebnis des direkt vorhergehenden Schrittes) auf dem Stack.

Anschließend wird eine Standard-API-Methode (die Methode [valueOf](#)) aufgerufen, welche den auf dem Stack übergebenen `int`-Parameter in eine Zeichenkette wandelt und die Referenz darauf als Rückgabewert auf dem Stack platziert.

Dieser Rückgabewert wird in der fünften lokalen Variable (Indexnummer 4) abgelegt.

Anschließend werden die in zwischenzeitlich den vierten und fünften lokalen Variablen abgelegten Adressen des Ausgabe-Streams und der auszugebenden Zeichenkette geladen und auf dem Operanden-Stack abgelegt.

Durch Aufruf der Standard-Ausgabemethode [println](#) mittels [invokevirtual](#) wird die referenzierte Zeichenkette auf der Standardausgabe dargestellt.

Der folgende Anweisungsblock demonstriert eine Ganzzahldivision mittels `idiv` welche Divisor und Dividenten als Operanden auf dem Stack erwartet und das Berechnungsergebnis ebenda platziert.

Anschließend wird das (noch auf dem Stack liegende) Berechnungsergebnis direkt weiterverarbeitet und in eine Zeichenkette gewandelt. Hierbei kommt die bereits bekannte Funktion zum Einsatz.

Danach erfolgt wiederum die Ausgabe in der bekannten Form.

Abschließend wird eine fixe Zeichenkette ausgegeben, deren Zeichenkettendarstellung nicht berechnet zu werden braucht. Ihr Wert kann daher direkt aus dem Laufzeitkonstantenpool per [ldc](#) geladen werden.

Die übrigen Schritte zur Erzeugung der Ausgabe bleiben indes unverändert.

Nutzung von Methoden:

Bereits bei den einfachen Operationen aus Beispiel 1 zeigt sich, daß die wiederholte Angabe von sehr ähnlichen Instruktionsfolgen nicht zu vermeiden ist. Insbesondere die beiden Konversionen des `int`-Datentyps als Voraussetzung der zeichenbasierten Ausgabe ist vollständig identisch.

Zur Strukturierung stehen daher auf der Java-Assemblerebene die bereits aus der Java-Hochsprache bekannten *Methoden* zur Verfügung, wie Beispiel 2 zeigt.

Beispiel 2: Nutzun von Methoden

```
(1).class public examples/BC2
(2).super java/lang/Object
(3)
(4).method public <init>()V
(5)    aload_0
(6)    invokevirtual java/lang/Object/<init>()V
(7)    return
(8).end method
(9)
(10).method public static printInt(I)V
(11)    .limit locals 2
(12)    .limit stack 2
(13)
(14)    iload_0
(15)    invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
(16)    astore_1
(17)
(18)    getstatic java/lang/System/out Ljava/io/PrintStream;
(19)    aload_1
(20)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(21)    return
(22).end method
(23)
(24).method public static main([Ljava/lang/String;)V
(25)    .limit locals 50
(26)    .limit stack 40
(27)
(28)    iconst_2
(29)    istore_0
(30)
(31)    bipush 101
(32)    istore_1
```



```

(33)
(34)    bipush 99
(35)    istore_2
(36)
(37)    ;we will need this twice
(38)    getstatic java/lang/System/out Ljava/io/PrintStream;
(39)    astore_3
(40)
(41)    iload_1
(42)    iload_2
(43)    iadd
(44)    istore_1
(45)
(46)    iload_1
(47)    invokestatic examples/BC2/printInt(I)V
(48)
(49)    iload_1
(50)    iload_0
(51)    idiv
(52)
(53)    invokestatic examples/BC2/printInt(I)V
(54)
(55)    ;print a fixed string
(56)    aload_3
(57)    ldc "The End"
(58)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(59)    return
(60).end method

```

[Download des Beispiels](#)

Die Funktionalität des Beispiels ist mit der der vorhergehend vorgestellten Codesequenz identisch. Jedoch finden sich jetzt die Instruktionsfolgen zur Berechnung der Zeichenkettenrepräsentation einer Ganzzahl und ihrer anschließenden Ausgabe in die Methode `printInt` ausgelagert.

Diese Methode akzeptiert eine Ausprägung des Primitivtyps `int` als Übergabe und liefert keinen Rückgabewert. Die Signatur ist daher dahingehend vereinbart, daß genau eine `int`-konforme Zahl als Parameter auf dem Stack erwartet wird, d.h. der Aufrufer hat diese vor dem Aufruf dort abzulegen.

Zusätzlich benötigt die Methode selbst zu ihrer Ausführung einige lokale Variablen, die auf dem methodenspezifischen Stack abgelegt werden. Dieser stellt eine Erweiterung des bereits durch den Aufruf verwendeten Operandenstacks dar.

Mit Java steht jedoch keineswegs die einzige Hochsprache zur Erzeugung von Byte-Code-Dateien zur Verfügung. [Diese Seite](#) listet eine Vielzahl verschiedener Alternativen.

Beispielsweise erzeugt der [Oberon-Compiler von Canterbury](#) für alle Oberon-Module, einschließlich der Systemmodule, Java-Klassen.

Beispiel 3: Die Hello World Applikation als Oberon Programm



```

MODULE helloworld;

IMPORT Out;

BEGIN
    Out.String( "Hello World" );
    Out.Ln;
END helloworld.

```

[Download des Beispiels](#)

Die erzeugten class-Dateien -- [SYSTEM.class](#), [helloworld.class](#), [Out.class](#), [Sys.class](#) -- können auf jeder JVM zur Ausführung gebracht werden. `java helloworld` liefert das erwartete Ergebnis.

Das Class-File-Format

Ausgangspunkt jeder Programmausführung innerhalb der JVM ist die class-Datei als Eingabe. Sie wird üblicherweise durch den Java-Compiler (im JDK: [javac](#) erzeugt).

Einige Eigenschaften jedes class-Files:

- Es enthält die Definition genau einer Klasse oder einer Schnittstelle, unabhängig von deren Zugriffs- und Sichtbarkeitseigenschaften.
 - Es wird als Bytestrom aufgefaßt, daher werden alle größeren Informationseinheiten durch serielles Lesen und aneinanderfügen gebildet.
- Unabhängig von der physischen Realisierung der tatsächlich zugrundeliegenden Hardware werden alle Multibyte-Daten im *big endian*-Format, d.h. größere Einheiten werden durch Linkshift und Boole'sches ODER gebildet, abgelegt.

Die Schnittstellen [java.io.DataInput](#), [java.io.DataOutput](#), [java.io.DataInputStream](#) und [java.io.DataOutputStream](#) unterstützen dieses Format. ([Beispiel](#))

- Die Strukturen innerhalb der class-Datei werden nicht zusätzlich optimiert abgelegt, daher erfolgt weder ein Auffüllen auf spezifische Wortgrenzen, noch ein Alignment an solchen.

Die JVM-Spezifikation legt zur Definition der Struktur des class-Files [eigene Datentypen](#) fest: u1, u2 und u4 zur Definition vorzeichenloser ein-, zwei- und drei-Bytetypen. Für diese (von der Java-üblichen vorzeichenbehafteten Mimik (abgesehen von char) abweichenden) Datentypen stehen mit [readUnsignedByte\(\)](#), [readUnsignedShort\(\)](#) und [readInt\(\)](#) entsprechende Lesemethoden zur Verfügung.

[The class File Format @ Java Virtual Machine Specification](#)

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

[Constant Pool @ Java Virtual Machine Specification](#)

```
cp_info {
    u1 tag;
    u1 info[];
}
```

[field_info @ Java Virtual Machine Specification](#)

```
field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

[method_info @ Java Virtual Machine Specification](#)

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

[attribute_info @ Java Virtual Machine Specification](#)

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

Aufbau einer class-Datei verdeutlicht an nachfolgendem Beispiel Quellcode.

Hinweis: Mit [Classeditor](#) existiert ein freies Werkzeug zur Inspektion und Modifikation übersetzter Class-Dateien.

Beispiel 4: Java-Quellcode der untersuchten Klassendatei

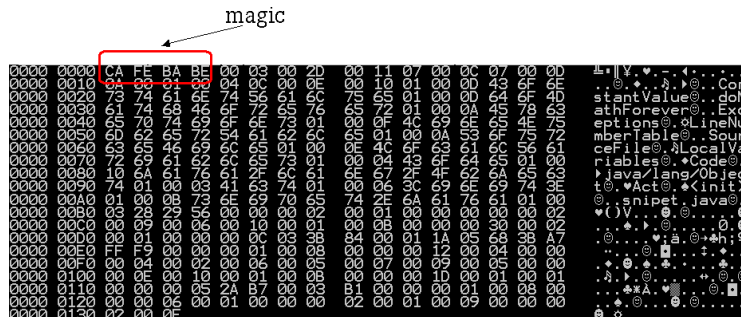


```

(1)class Act {
(2)    public static void doMathForever() {
(3)        int i=0;
(4)        while (true) {
(5)            i += 1;
(6)            i *= 2;
(7)        } //while
(8)    } //doMathForever()
(9)} //class Act

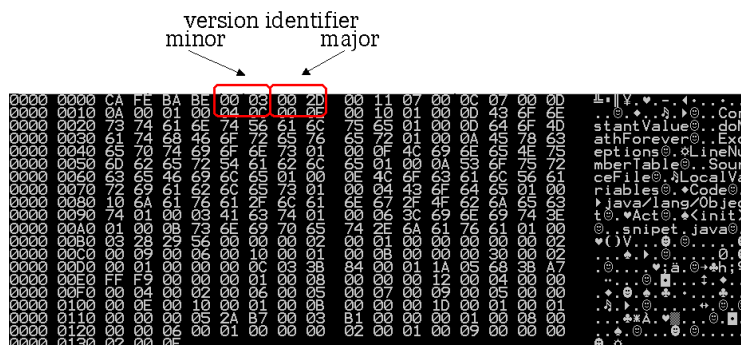
```

Download des Beispiels



Der magic-Identifier ist auf die Bytekombination (in hexadezimaler Darstellung) CA FE BA BE fixiert. Anhand dieser erkennt der Kassenlader der Laufzeitsystems die Datei als ausführbare Java-Bytecode-Datei an.

Ist diese gegenüber dem Vorgabewert modifiziert wird eine `java.lang.ClassFormatError` Ausnahme im Hauptthread generiert (Bad magic number wird als zusätzliche Nachricht der Ausnahme ausgegeben). [siehe JVM-Spezifikation](#)



Die beiden Versionskennungen minor und major bilden gemeinsam den Versionsschlüssel der class-Datei in der gängigen Punktnotation. Hierbei gilt: major.minor

Die Klassendatei des Beispiels trägt den Versionsschlüssel 45.3.

Der im SUN JDK enthaltene Java-Compiler erlaubt per Kommandozeilenparameter (target) die JVM-spezifische Steuerung der Codegenerierung. Die den einzelnen Sprachversionen zugeordneten Bytecodeversionen sind in der nachfolgenden Tabelle zusammengestellt.

Java-Version	Bytecode-Version
1.1	45.3
1.2	46.0
1.3	47.0
1.4 (sowie 1.4.1, 1.4.2 und der Prototyp des 1.5-Compilers)	48.0
Zweite Vorabversion des 1.5-Compilers	50.0
1.5 (beta1)	49.0
1.5 (beta2)	49.29

Vorgabegemäß wird durch die Compilerversion 1.5 (ab Beta-Version 2) 49.29 erzeugt.

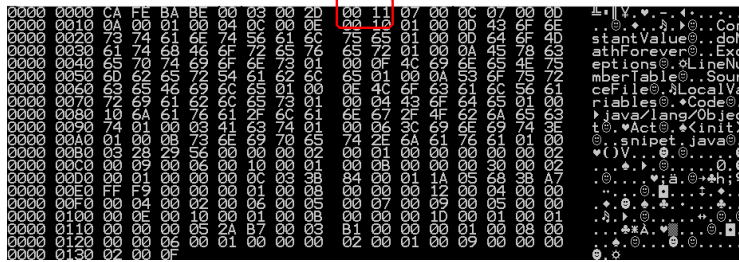
Trägt eine class-Datei eine durch die JVM nicht unterstützte Versionsnummer, so wird eine `java.lang.UnsupportedClassVersionError`-Ausnahme generiert.

Jede JVM kann verschiedene class-Datei-Versionen unterstützen, die letztendlich Festlegung welche Versionen durch einzelne Java-Plattform-Releases zu unterstützen sind obliegt jedoch SUN. So unterstützt SUNs JDK v1.0.2 class-Dateien der Versionen 45.0 bis einschließlich 45.3. Die JDK-Generation v1.1.x ab Version 45.0 bis einschließlich 45.65535 und Implementierungen der Java 2 Plattform, Version 1.2, bis einschließlich 46.0. JDK v1.3.0 verarbeitet Klassendateien bis hin zur Versionsnummer 47.0. Zur Verarbeitung von Klassen, welche die in 1.5 eingeführten Generizitätsmechanismen verwenden nicht zwingend eine Ausführungsumgebung dieser Versionsnummer benötigt, da das erzeugte Klassenformat (bisher, da diese Aussagen auf dem Informationsstand der verfügbaren Betaversion basieren) nicht verändert wurde. Die Nutzung des dynamischen Boxing/Unboxings benötigt jedoch eine

Ausführungsumgebung mindestens der Version 1.5.

[siehe JVM-Spezifikation](#)

constant pool count



Die Bytefolge `constant_pool_count` enthält die um eins erhöhte Anzahl der Einträge der `constant_pool` Tabelle.

Im Beispiel ist dies: 17.

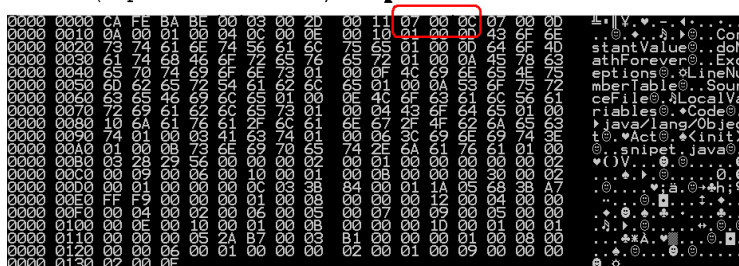
[siehe JVM-Spezifikation](#)

Der `constant pool` enthält die symbolische Information über Klassen, Schnittstellen, Objekte und Arrays. Die Elemente dieser Datenstruktur sind vom Typ `cp_info` und variieren je nach tag in ihrer Länge. Als Konstantentypen (= Inhalt des Tag-Bytes) sind zugelassen:

Konstantentyp	Wert
CONSTANT_Utf8	1
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_NameAndType	12
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_CLASS	7
CONSTANT_String	8
CONSTANT_Fieldref	9

[siehe JVM-Spezifikation](#)

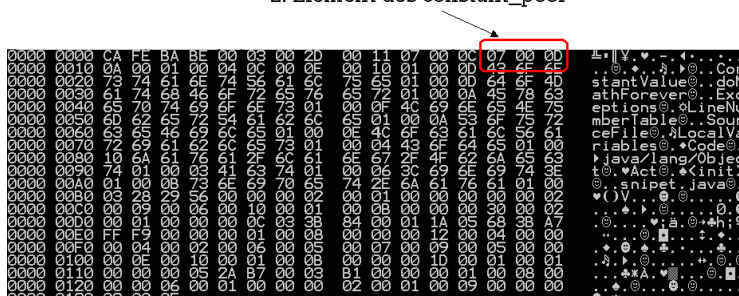
1. Element des `constant_pool`
(Superklassen-Verweis)



Konstante vom Typ `CONSTANT_class` die einen Verweis auf auf das zwölfte Element des Konstantenpools (0x0C) enthält. Zum Lesezeitpunkt der ersten Konstante kann diese Referenz noch nicht aufgelöst und auf Gültigkeit geprüft werden. (Später sehen wir, daß es sich um eine Referenz auf `java.lang.Object` handelt).

Hierbei handelt es sich immer um die Referenz auf die Superklasse. Auch für die API-Klasse `Object` selbst findet sich diese Referenz in der Klassendatei, auch wenn Diassemblierungswerkzeuge wie `javap` diese nicht ausgeben.

2. Element des `constant_pool`



[illegible][illegible][illegible][illegible]

<http://www.ieckle.de/vorlesung/java/script.html> (68 of 135)08.06.2004 21:43:34

7. Element des constant_pool (Zeichenkette "Exceptions")

A hex dump showing the memory layout of the constant pool entry for the string "Exceptions". The entry is of type CONSTANT_Utf8. The first two bytes are 00 00, indicating the type. The next four bytes are CA FE BA BE, which are the UTF-8 bytes for the string "Exceptions". The rest of the entry is padding (00 00).

Konstante vom Typ CONSTANT_Utf8, die den fixen String *Exceptions* einleitet.

8. Element des constant_pool (Zeichenkette "LineNumberTable")

A hex dump showing the memory layout of the constant pool entry for the string "LineNumberTable". The entry is of type CONSTANT_Utf8. The first two bytes are 00 00, indicating the type. The next four bytes are CA FE BA BE, which are the UTF-8 bytes for the string "LineNumberTable". The rest of the entry is padding (00 00).

Konstante vom Typ CONSTANT_Utf8, die den fixen String *LineNumberTable* einleitet.

9. Element des constant_pool (Zeichenkette "SourceFile")

A hex dump showing the memory layout of the constant pool entry for the string "SourceFile". The entry is of type CONSTANT_Utf8. The first two bytes are 00 00, indicating the type. The next four bytes are CA FE BA BE, which are the UTF-8 bytes for the string "SourceFile". The rest of the entry is padding (00 00).

Konstante vom Typ CONSTANT_Utf8, die den fixen String *SourceFile* einleitet.

10. Element des constant_pool (Zeichenkette "LocalVariable")

A hex dump showing the memory layout of the constant pool entry for the string "LocalVariable". The entry is of type CONSTANT_Utf8. The first two bytes are 00 00, indicating the type. The next four bytes are CA FE BA BE, which are the UTF-8 bytes for the string "LocalVariable". The rest of the entry is padding (00 00).

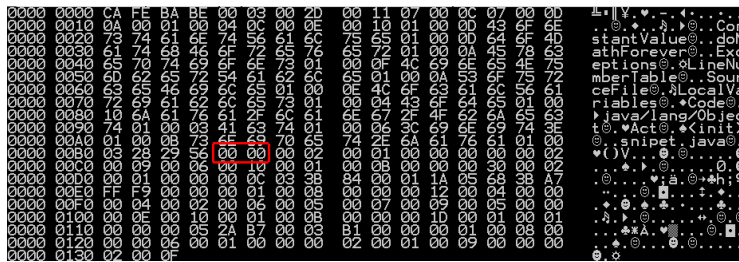
Konstante vom Typ CONSTANT_Utf8, die den fixen String *LocalVariables* einleitet.

11. Element des constant_pool (Zeichenkette "Code")

A hex dump showing the memory layout of the constant pool entry for the string "Code". The entry is of type CONSTANT_Utf8. The first two bytes are 00 00, indicating the type. The next four bytes are CA FE BA BE, which are the UTF-8 bytes for the string "Code". The rest of the entry is padding (00 00).

Konstante vom Typ CONSTANT_Utf8, die den fixen String *Code* einleitet.

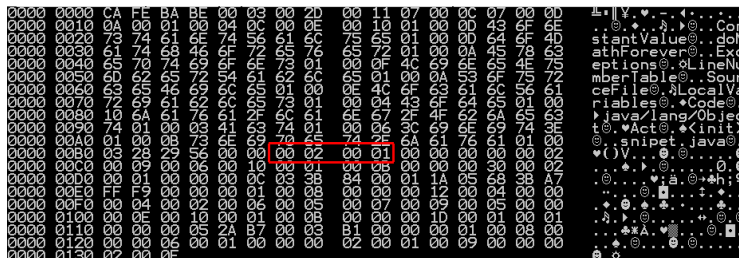
Zugriffsflags



Zugriffsflags für die Klasse Act. Der konkrete Code ergibt sich aus der binären ODER-Verknüpfung verschiedener Zugriffsflaggen, die in untenstehender Tabelle wiedergegeben sind.

Flag	Wert	Beschreibung
ACC_PUBLIC	0x0001	public-Deklaration; Zugriffbar von allen anderen Klassen, auch außerhalb des eigenen Pakets
ACC_FINAL	0x0010	final-Deklaration; Verbot der Vererbung
ACC_SUPER	0x0020	Besondere Behandlung der Superklasseninstruktionen bei Aufruf über Opcode invokespecial
ACC_INTERFACE	0x0200	Struktur ist Schnittstelle, keine Klasse
ACC_ABSTRACT	0x0400	Abstrakte Struktur, von der keine Ausprägungen erzeugt werden können

Referenz auf this und super

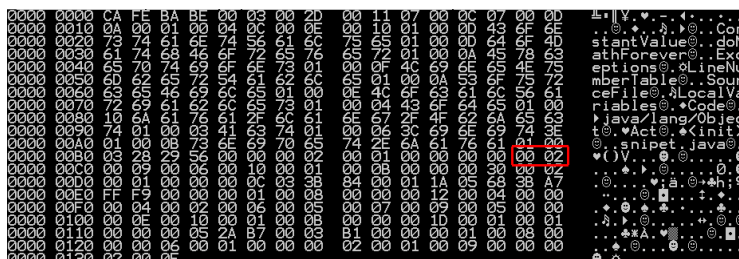


Referenz in den Konstantenpool, auf die Klasse selbst (this) und die Superklasse (super).

interface_count und
fields_count

interface_count: Anzahl der durch die Klasse implementierten Schnittstellen.
fields_count: Anzahl der Klassen- oder Instanzvariablen der Klasse.

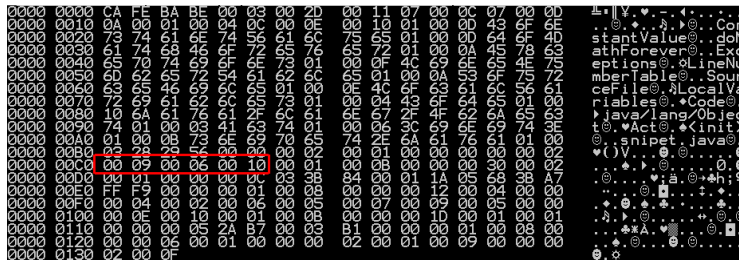
methods_count



Anzahl der durch die Klasse implementierten Methoden.

Die Zahl ergibt sich aus der tatsächlich durch den Anwender definierten Anzahl und den impliziten, d.h. durch den Compiler hinzugefügten (z.B. Konstruktor), Methoden.

methods_info (doMathForever())



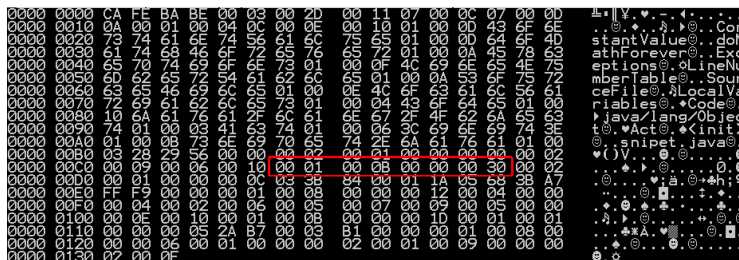
Informationen über die in der Klasse Act definierten Methoden.

Die Zugriffsflaggen (0x00 09) weisen doMathForever() als static und public aus. (Die konkrete Wertebelegung kann untenstehender Aufstellung entnommen werden. Auch in diesem Falle wird der tatsächliche Wert durch Boole'sche ODER-Verknüpfung der Einzelwerte gebildet.)

Der Verweis (0x06) auf das sechste Element des Konstantenpools referenziert den Methodennamen im Klartext. Der zweite Verweis (0x10) auf den Konstantenpool kennzeichnet doMathForever() als parameterlose Methode ohne Rückgabetypen.

Flag	Wert	Beschreibung
ACC_PUBLIC	0x0001	public-Deklaration; Zugriffbar von allen anderen Klassen, auch außerhalb des eigenen Pakets
ACC_PRIVATE	0x0002	Als private deklariert, daher nur innerhalb der definierenden Klasse verwendbar
ACC_PROTECTED	0x0004	protected-Deklaration, Zugriff nur in Subklassen möglich
ACC_STATIC	0x0008	static-Deklaration, keine Auswirkungen auf Sichtbarkeit und Zugriffsrechte
ACC_FINAL	0x0010	final-Deklaration; nach initialer Zuweisung keine Wertänderung möglich
ACC_VOLATILE	0x0040	volatile-Deklaration; keine Berücksichtigung in Optimierungsmaßnahmen
ACC_TRANSIENT	0x0080	transient-Deklaration; keine Berücksichtigung durch Persistenzmanager

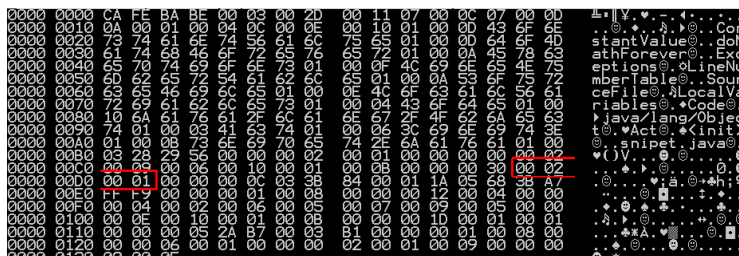
Attribute der Methode doMathForever()



Die Methode doMathForever() verfügt nur über genau ein Attribut, daher ist der attribute count zu Beginn der Bytesequenz auf 0x00 01 gesetzt. Dieses eine Attribut wird durch Index 11 innerhalb des Konstantenpools referenziert. Dort ist die Zeichenkette Code lokalisiert. Dadurch wird angezeigt, daß die folgenden Bytes die Implementierung dieser Methode beinhalten.

Der abschließende vier-Byte Indikator enthält die Länge der Methodenimplementierung (im Beispiel: 0x30).

maxStack und maxLocals der Methode doMathForever()



maxStack: Maximalhöhe des Operandenstacks die während der Methodenausführung erreicht werden kann. (Im Beispiel: 2; die Opcode-Implementierung der beiden verwendeten arithmetischen Operationen benötigen niemals mehr als zwei Stackpositionen.)

maxLocals: Anzahl der lokalen Variablen. (die verwendete Variable i)

Bytecode und Ausnahmentabelle der Methode doMathForever()

Die code length legt die Anzahl der folgenden Bytecode-Instruktionen fest (im Beispiel: 12), darauf folgen die tatsächlichen Opcodes.

pc	instruction	mnemonic
0	03	iconst_0
1	3B	istore_0
2	840001	iinc 0 1
5	1A	iload_0
6	05	iconst_2
7	68	imul
8	3B	istore_0
9	A7FFF9	goto 2

Die Ausnahmentabelle (Exception Table) enthält die Anzahl der durch die Methode aufgefangenen Ausnahmeereignisse; im Beispiel: 0.

Eigenschaften des Code-Bereichs der Methode doMathForever()

In diesem Bereich werden zusätzliche Charakteristika des bereits definierten Codebereichs hinterlegt, z.B. Debugginginformation.

Im betrachteten Falle ist nur eine Eigenschaft angegeben (attribute_count = 0x01). Diese referenziert das achte Element des Konstantenpools -- die Zeichenkette LineNumberTable. Die beiden abschließenden Attribute bezeichnen die Länge dieser Tabelle (0x12) und die Anzahl der Einträge (0x4).

Die LineNumberTable des Beispiels:

```
line 4: i = 0;
line 5: while(true) {
line 6: i += 1;
line 7: i *= 2;
```

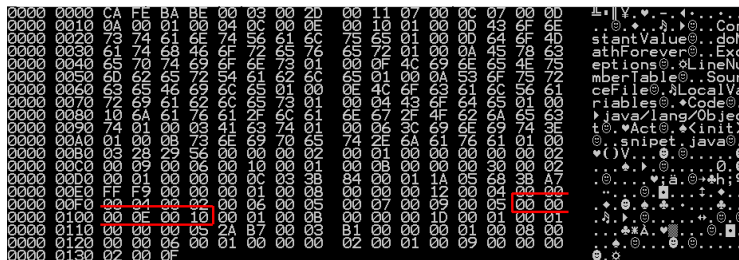
Zuordnung zwischen LineNumberTable und der Methode doMathForever()

Diese Datenstruktur stellt die Zuordnung zwischen den Quellcodezeilen und den resultierenden Opcodes her.

```
LineNumberTable[0]: iconst_0    istore_0
LineNumberTable[1]: iinc 0 1
LineNumberTable[2]: iload_0     iconst_2    imul    istore_0
LineNumberTable[3]: goto 2
```

Zugriffsflaggen und Indizes der Klasse

Act()



Diese zweite methodenbezogene Struktur gibt Auskunft über den Konstruktor der Klasse.

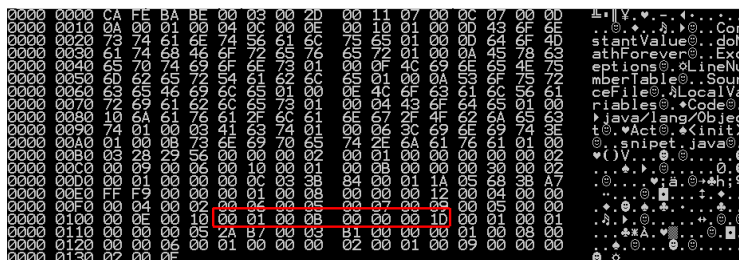
Im ersten Doppelbyte sind die Zugriffsrechte spezifiziert; in diesem Falle sind keine gesonderten Festlegungen getroffen -- es handelt sich um eine *einfache* Methode.

Die Referenz in den Konstantenpool verweist auf die implementierende Methode (im Beispiel: Position 0x0E, dort findet sich die Methode <init>).

Durch die letzten beiden Bytes wird der Typ des Konstruktors referenziert, im betrachteten Beispiel die Position 0x10 im Konstantenpool, mithin ein parameterloser Konstruktor.

Attribute der Klasse

Act()

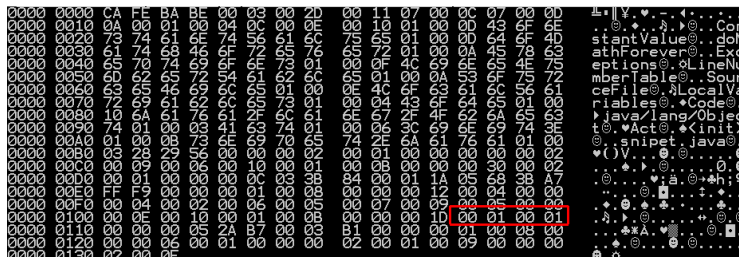


Der Zähler (ersten beiden Bytes) zu Beginn der Struktur zeigt an, daß nur ein Attribut der Klasse Act() folgt. Im Beispielfall handelt es sich dabei um das über den Index 11 (0x0B) angesprochene Element des Konstantenpools, die Zeichenkette Code.

Der abschließend angegebene Längenzähler fixiert die Anzahl der folgenden Bytes.

maxStack und maxLocals der Klasse

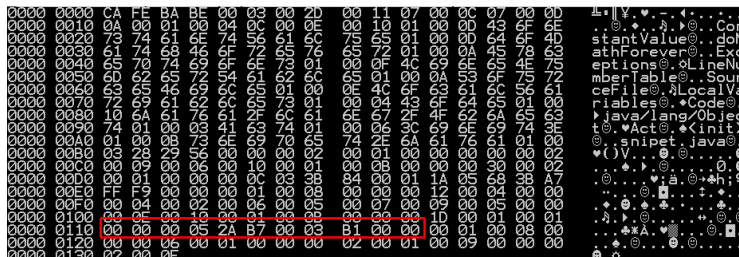
Act()



Analog der Definition für Methoden, die maximale Höhe des Operandenstacks und die Anzahl der lokalen Variablen.

Bytecode und Ausnahmetabelle der Klasse

Act()



Opcode-Implementierung des Konstruktors, sowie die Aufzählung der durch ihn potentiell ausgelösten Ausnahmeereignisse (im keine, daher Anzahl gleich Null).

Die Implementierung in Java-Bytecode:

pc	instruction	mnemonic
0	2A	aload_0
1	B70003	invokeonvirtual #3 <Method java.lang.Object <init> ()V>
4	B1	return

puffert ihn zwischen. Bei jeder erneuten Ausführung derselben Bytecodesequenz wird nun dieser bereits übersetzte Code ausgeführt. Dieses Vorgehen ist in den verbreiteten JVM-Implementierungen der Internet-Browser von Netscape und Microsoft verwirklicht. Ebenso bieten fast alle verfügbaren Java-Entwicklungsumgebungen diese Laufzeit-optimierende Komponente an. Der dadurch erzielte Geschwindigkeitsvorteil bewegt sich, je nach Struktur der Anwendung, zwischen zehn und 50 Prozent.

Den größten Geschwindigkeitszuwachs verspricht man sich jedoch von der vollständigen Realisierung der virtuellen Maschine in Hardware; damit wird sie de facto zur realen Maschine. Hierzu liegen jedoch noch keine Ergebnisse vor, welche die der derzeitigen Implementierung auf handelsüblichen Prozessoren signifikant überträfen.

Eine andere Sichtweise nutzt das Bytecodeformat welches eine Zwischenrepräsentation darstellt nicht zur Interpretation mit dem Ziele der direkten Ausführung, sondern als Eingabeformat eines weiteren Übersetzungsschrittes, der üblicherweise plattformabhängigen nativen Code erzeugt. Für C++ existieren bereits Umsetzungen, die Bytecode in übersetzungsfähigen C++-Quellcode transformieren. Eine Spielart hiervon bildet das diskutierte Werkzeug javap dessen Ausgabeformat, nach einigen Umformungen, direkt als Eingabe weiterer Übersetzer akzeptiert wird.



Web-Referenzen 1: Weiterführende Links

- [Inside the Java Virtual Machine](#)

3.2 Die Java API und weiterführende Themen

3.2.1 Ein- / Ausgabe -- Streams

Zur Erinnerung: **bisher behandelte Möglichkeiten zur Ein- und Ausgabe:**

Bisher wurden ausschließlich Bildschirm-Ausgaben, und diese ausschließlich mit der statischen Methode `System.out.println`, erzeugt werden. Der einzige uns bisher bekannte Mechanismus zur Eingabebehandlung war der der Kommandozeilenparameter in der `main`-Methode.

Wie aus C++ bekannt verfügt auch Java über die objektorientierte Kapselung der Ein- und Ausgabebehandlung in Form von *Streams*. Hierbei stehen beliebigste Ein- und Ausgabequellen über denselben programmiersprachlichen Mechanismus zur Verfügung, unabhängig davon wie das physische Gerät realisiert ist.

Zentrales Paket für die Ein-/Ausgabebehandlung ist [java.io](#). Es enthält neben den wichtigsten Klassen zur Implementierung des E/A-Verhaltens auch verschiedene Schnittstellen, sowie die möglichen Ausnahmeereignisse während der verschiedenen E/A-Operationen.

Gegenüber den aus C++ bekannten Datenströmen tritt bei Java hinzu, daß auch Informationen über Netze mit denselben Mechanismen übertragen werden.

In Java werden generelle zwei Streamtypen unterschieden: **Character Streams** und **Byte Streams**. Wie der Name schon andeutet, sind Byte Streams auf die Verarbeitung von beliebigen Byte-artigen Informationseinheiten -- und damit acht Bit große Einheiten -- beschränkt. Diese Mimik stellt insbesondere bei der Verarbeitung von Unicode-Zeichenketten eine große Einschränkung dar, da hierbei nicht gesamte Zeichen-Information (ein Symbol mißt 16 Bit) in einem Zugriff verarbeitet werden kann. Daher wurde mit dem JDK v1.1 zusätzlich das Konzept der Character Streams eingeführt, die generell 16 Bit lange Zeichen bereitstellen.

Als **Byte Streams** stehen zur Verfügung:

(Einrückungen kennzeichnen Subklassenbeziehungen, Kursivsetzungen abstrakte Klassen)

- [InputStream](#)
Abstrakte Basisklasse der meisten Eingabeströme. Sie definiert die Signatur die [grundlegenden Eingabemethoden](#).
 - [FileInputStream](#)
Byteweises Lesen einer Datei.
 - [PipedInputStream](#)
Hochsprachliches Äquivalent der Betriebssystem Pipes. Einsatzfeld: Gemeinsam mit [PipedOutputStream](#) zum Datenaustausch zwischen verschiedenen [Threads](#) derselben Applikation.
 - [FilterInputStream](#)
Gemeinsam mit anderen Eingabeströmen eingesetzt. Einsatzgebiet: Nachbearbeitung roher Eingaben. Die entsprechenden Subklassen stellen konkrete Implementierungen häufiger Einsatzfälle dar.
 - [LineNumberInputStream](#)*deprecated*
Liefert zur Eingabe die zugehörige Zeilennummer des Eingabestroms.
Achtung: Diese Klasse geht von der (i. A. falschen) Entsprechung zwischen Bytes und Charactern aus. [LineNumberReader](#) liefert für denselben Anwendungsfalle eine korrekte Implementierung.
 - [DataInputStream](#)
Liest [primitive Java-Datentypen](#) plattformunabhängig aus einem Datenstrom.
 - [BufferedInputStream](#)
Verleiht beliebigen [InputStreams](#) die Möglichkeit des gepufferten Lesens aus dem Eingabestrom. Zusätzlich wird das Setzen von, und die Rücksetzung des Strompositionszeigers zu, Markierungspunkten unterstützt.
 - [PushbackInputStream](#)
Verleiht beliebigen [InputStreams](#) die Möglichkeit der Wiedereinstellung von bereits gelesenen Byte-Inhalten in den Eingabestrom.
 - [ByteArrayInputStream](#)
Lesen aus einem Byte Array.
 - [StringBufferInputStream](#)*deprecated* Lesen aus String.*Achtung:* Diese Klasse geht von der (i. A. falschen) Entsprechung zwischen Bytes und Charactern aus. [StringReader](#) liefert für denselben Anwendungsfalle eine

- korrekte Implementierung.
 - [SequenceInputStream](#)
Konkatenation von Eingabströmen; nach vollständigem Lesen des i -ten Eingabstroms wird der $i+1$ -te angesprochen.
 - [ObjectInputStream](#)
Lesen vollständig serialisierter Objekte, die die Schnittstellen [java.io.Serializable](#) oder [java.io.Externalizable](#) unterstützen.
- [OutputStream](#)
Abstrakte Basisklasse der meisten Ausgabeströme. Sie definiert die Signatur der [grundlegenden Ausgabemethoden](#).
 - [FileOutputStream](#)
Schreibt Byte-artige Daten in Dateien oder durch Dateideskriptoren bezeichnete Ziele.
 - [PipedOutputStream](#)
Umkehrung (Schreibende) eines [PipedInputStreams](#).
 - [FilterOutputStream](#)
Verleiht Ausgabeströmen die Möglichkeit Ausgabedaten zu verändern. Einige konkrete Implementierungen sind durch die Subklassen gegeben.
 - [DataOutputStream](#)
Erlaubt plattformunabhängiges Schreiben der [primitiven Java-Datentypen](#).
 - [BufferedOutputStream](#)
Verleiht verschiedenen [OutputStreams](#) die Möglichkeit des gepufferten (d.h. nicht mehr Byte-weisen) Schreibens.
 - [PrintStream](#)
Verleiht anderen [OutputStreams](#) diverse Möglichkeiten zur Ausgabe der verschiedenen Java Datentypen. Anders als die übrigen Stromtypen erzeugt PrintStream keine [IOException](#).
Anmerkung: Es werden Bytes, keine Characters geschrieben. Mit [PrintWriter](#) existiert eine Standardimplementierung, die auf die Besonderheiten im Zusammenhang mit Unicode-Ausgaben Rücksicht nimmt.
 - [ByteArrayOutputStream](#)
Ausgabestrom, der in einen Byte-Array schreibt.
 - [ObjectOutputStream](#)
Schreibt [Primitivtypen](#) und Objekte. Ausschließlich Objekte, welche die [java.io.Serializable](#)-Schnittstelle implementieren können über diesen Mechanismus persistent geschrieben werden.

Als **Character Streams** stehen zur Verfügung:

(Einkrückungen kennzeichnen Subklassenbeziehungen, Kursivsetzungen abstrakte Klassen)

- [Reader](#)
[Abstrakte Basisklasse der verschiedenen Reader-Implementierung. Sie definiert die Signatur der grundlegenden Character-bezogenen Ausgabemethoden.](#)
 - [BufferedReader](#)
Liest Textinhalte gepuffert aus einem Character-Eingabestrom.
 - [LineNumberReader](#)
Liest einzelne Unicode-Zeichen, Zeichensequenzen und ganze Zeilen gepuffert, und stellt zusätzlich die Zeilennummer des aktuell verarbeiteten Inhaltes.
 - [CharArrayReader](#)
Einlesen von Unicode-Zeichen aus Character-Puffer.
 - [InputStreamReader](#)
 - [FileReader](#)
Eingabestrom für character-Dateien.
 - [FilterReader](#)
Abstrakte Basisklasse verschiedener filternder Eingabeströme. Konkrete Implementierungen werden durch die verschiedenen Subklassen zur Verfügung gestellt.
 - [PushbackReader](#)
Unicode-Character Eingabestrom, der die Rückstellung von bereits gelesenen Inhalten in den Strom erlaubt.
 - [PipedReader](#)
Unicode-Character Eingabestrom als Lese-Ende einer Pipe.
 - [StringReader](#)
Unicode-Character Eingabestrom, der aus einem [String](#) liest.
- [Writer](#)
Abstrakte Basisklasse verschiedener Implementierungen zur Ausgabe von Unicode-Zeichenströmen.
 - [BufferedWriter](#)
Gepuffertes Schreiben. Zeilenvorschub wird transparent plattformabhängig umgesetzt. (siehe [system properties](#)).
 - [CharArrayWriter](#)
 - [OutputStreamWriter](#)
Brücke zwischen Character und Byte-Strömen; Unicode-Zeichen werden automatisch während des Schreibens in die entsprechende Byte-Repräsentation umgesetzt.
 - [FileWriter](#)
Schreibt Character-Dateien.
 - [FilterWriter](#)
Abstrakte Basisklasse der verschiedenen Implementierungen zum gefilterten Schreiben.
 - [PipedWriter](#)
Ausgabestrom über Character-Pipe.
 - [StringWriter](#)
Ausgabestrom, der in einen [String](#) schreibt.
 - [PrintWriter](#)
Ausgabestrom der formatierte Objektrepräsentationen schreibt. Methoden dieser Klasse verursachen keine E/A-Ausnahmeereignisse.

Üblicherweise existieren die Ströme immer symmetrisch, d.h. in gleicher Weise sowohl für Ein- als auch Ausgabe. Dieses Prinzip wird nur für einzelne Klassen durchbrochen, die zwar Eingabeseitig existieren (beispielsweise Lesen mit Zeilennummer), für die jedoch kein expliziter Ausgabemechanismus benötigt wird.

Zusätzlich zu den nach Zugriffsarten klassifizierten Strömen existiert mit der Klasse [RandomAccessFile](#) eine Strom über den sowohl lesende als auch schreibende Zugriffe abgewickelt werden können.

Die abstrakten Basisklassen [InputStream](#) bzw. [Reader](#) und definieren ähnliche Lese- und Zugriffsmethoden, die in allen abgeleiteten Klassen auf den entsprechenden Stromtypen zur Verfügung stehen.

Die aktuell gewünschte Zugriffsart (nur-lesend, nur-schreibend oder beides) wird über einen Parameter des Konstruktors gesteuert.

Wie durch den Klassennamen bereits angedeutet, existiert dieser Strom ausschließlich für Dateien; eine Netzwerkanwendung ist nicht möglich.

grundlegende Lesemethoden:

- [java.io.InputStream](#):
 - [int available\(\)](#) Liefert die Anzahl Bytes die an der Eingabeschchnittstelle zur Verfügung stehen. Diese Anzahl kann ohne Blockierung des Aufrufers gelesen werden.
 - [void close\(\)](#) Schließt Eingabestrom unter Freigabe der belegten Systemressourcen
 - [void mark\(int\)](#) Markiert die gegenwärtige Position des Eingabezeigers im Eingabestrom; ein folgender Aufruf von [reset](#) setzt den Eingabezeiger wieder an die markierte Position zurück.
 - [boolean markSupported\(\)](#) Gibt Auskunft darüber, ob der Eingabestrom die Positionsmarkierung, und das Rücksetzen darauf, unterstützt.
 - [int read\(\)](#) Liest den nächsten Bytewert (>0 und <256) aus dem Eingabestrom. Ist das Ende des Eingabestromes erreicht, wird -1 retourniert (kein Ausnahmeereignis! Die Verwendung des [StreamTokenizers](#) ermöglicht hier ein handlicheres Vorgehen).
Subklassen von [InputStream](#) sind gezwungen diese Methode zu überschreiben.
 - [int read\(byte\[\]\)](#) Liest eine Bytesequenz aus dem Eingabestrom, und legt sie im übergebenen Array ab. Die Anzahl der gelesenen Zeichen, bzw. -1 beim Erreichen den Eingabeendes, wird zurückgegeben.
 - [int read\(byte\[\] b, int off, int len\)](#) Liest Bytesequenz der Länge len und speichert sie ab Position off im übergebenen Array.
 - [void reset\(\)](#) Setzt den Stromzeiger auf die Position der letzten vorhergehenden Markierung zurück, falls eine solche existiert.
 - [long skip\(long n\)](#) Versucht den Stromzeiger um n Bytepositionen vorzurücken. Die Anzahl der tatsächlich übersprungenen Bytes wird zurückgegeben.
- [java.io.Reader](#):
 - [void close\(\)](#) Schließt den Ausgabestrom.
 - [void mark\(int readAheadLimit\)](#) Markiert gegenwärtige Position des Eingabezeigers. Ein nachfolgender Aufruf von [reset](#) versucht den Positionszeiger auf die Markierte Stelle zurückzusetzen, falls die aktuelle Position nicht weiter als [readAheadLimit](#) Zeichen entfernt liegt.
 - [void markSupported\(\)](#) Gibt Auskunft darüber, ob der Eingabestrom die Positionsmarkierung, und das Rücksetzen darauf, unterstützt.
 - [int read\(\)](#) zur Extraktion genau eines Zeichens (16 Bit Character).
 - [int read\(char\[\] cbuf\)](#) zur Extraktion einer Sequenz, beginnend ab der aktuellen Stromzeigerposition.
 - [int read\(char\[\] cbuf, int off, int len\)](#) zur Extraktion einer Sequenz der Länge len oder weniger von Zeichen, und Ablage in Array beginnend ab der Position off.
 - [boolean ready\(\)](#) Liefert dieser Aufruf true zurück, so liegen weitere Eingaben vor, die durch ein folgendes [read](#) gelesen werden können.
 - [reset\(\)](#) Versucht den Positionszeiger auf die durch die letzte gesetzte Markierung bezeichnete Stelle zurückzurücken.
 - [long skip\(long n\)](#) Versucht n Zeichenpositionen zu überspringen. Die Anzahl der tatsächlich übersprungenen Positionen wird zurückgeliefert.

grundlegende Schreibmethoden:

- [java.io.OutputStream](#):
 - [void close\(\)](#) Schließt Ausgabestrom und gibt durch ihn belegte Systemressourcen frei.
 - [void flush\(\)](#) Leert Ausgabestrom und schreibt alle Pufferbereiche.
 - [void write\(byte\[\]\)](#) Schreibt Byte-Array in Ausgabestrom.
 - [void write\(byte\[\] b, int off, int len\)](#) Schreibt len Bytes eines Byte-Arrays ab Position off.
 - [void write\(int\)](#) Schreibt Bytewert in Ausgabestrom.
Subklassen von [OutputStream](#) müssen diese dieser Methode überschreiben.
- [java.io.Writer](#):
 - [void close\(\)](#) Schließt Ausgabestrom nach erfolgter Pufferleerung (flushing).
Subklassen von [Writer](#) müssen diese Methode überschreiben.
 - [void flush\(\)](#) Leert Pufferbereiche.
 - [write\(int\)](#) Schreibt ein Unicode-Zeichen.
 - [write\(char\[\]\)](#) Schreibt Character-Array.
 - [write\(char\[\] cbuf, int off, int len\)](#) Schreibt Teil der Länge len eines Character-Arrays beginnend ab Position off.
 - [write\(String\)](#) Schreibt [String](#).
 - [write\(String str, int off, int len\)](#) Schreibt Teil der Länge len eines [Strings](#) beginnend ab Position off.

Mit den [Dateideskriptoren](#) [in](#), [out](#) und [err](#) stehen die aus C/C++ bekannten drei **Standardströme** zur Verfügung. Diese standardmäßig geöffneten Ströme stehen während der Ausführungszeit jeder Applikation zur Verfügung.

```

(1)import java.io.FileDescriptor;
(2)import java.io.FileWriter;
(3)import java.io.IOException;
(4)
(5)public class PrintLn {
(6)    public static void main(String[] args) {
(7)        FileWriter fw = null;
(8)        try {
(9)            fw = new FileWriter(FileDescriptor.out);
(10)           for (int i=0; i < args.length; i++)
(11)               fw.write (args[i]+" ");
(12)        } catch (IOException ioe) {
(13)            System.out.println("cannot open stdout!");
(14)        } finally {
(15)            try {
(16)                fw.close();
(17)            } catch (Exception e) {
(18)                //ignore it
(19)            } //catch
(20)        } //finally
(21)    } //main()
(22)} //class PrintLn

```

Beispiel 57: Ausgabe auf standard out [PrintLn.java](#)

Das Programm öffnet erzeugt ein `FileWriter`-Objekt mit dem vorgegebenen Dateideskriptor `out`. Anschließend werden die Kommandozeilenparameter (allesamt Typ `String`) mit `write` ausgegeben. Alle Methoden der Klasse `FileWriter` können ein Ausnahmeereigniss vom Typ [IOException](#) erzeugen. Daher müssen Operationen auf dem erzeugten Ausgabestrom durch `try`-Blöcke abgesichert werden.

Wird als Ausgabekanal des `FileWriter`-Stroms auf eine physikalische Datei ausgerichtet, so muß lediglich die Konstruktoranweisung zu `fw = new FileWriter("myFile.asc")` modifiziert werden. Alle E/A-Klassen interpretieren die Pfadangabe relativ zum aktuellen Verzeichnis. Die Verzeichnisseparatoren variieren plattformabhängig. Verzeichnistrenner und aktueller Katlog können über die [system properties](#) ermittelt werden.

Stream erzeugung und mögliche Datenquellen aller (nicht als deprecated gekennzeichneten) Streamtypen:

Strom	Erzeugbar aus
<code>BufferedInputStream</code>	InputStream
<code>BufferedOutputStream</code>	OutputStream
<code>BufferedReader</code>	Reader
<code>BufferedWriter</code>	Writer
<code>ByteArrayInputStream</code>	Byte Array
<code>CharArrayReader</code>	Character Array
<code>DataInputStream</code>	InputStream Beispiel
<code>DataOutputStream</code>	OutputStream
<code>FileInputStream</code>	File -Objekt FileDescriptor (nur auf die Standardströme <code>stdin</code> , <code>stdout</code> , <code>stderr</code> anwendbar) String , der einen gültigen Pfad enthält
<code>FileOutputStream</code>	File -Objekt, FileDescriptor (nur auf die Standardströme <code>stdin</code> , <code>stdout</code> , <code>stderr</code> anwendbar) String , der einen gültigen Pfad enthält
<code>FileReader</code>	File -Objekt, FileDescriptor (nur auf die Standardströme <code>stdin</code> , <code>stdout</code> , <code>stderr</code> anwendbar) String , der einen gültigen Pfad enthält
<code>FileWriter</code>	File -Objekt, FileDescriptor (nur auf die Standardströme <code>stdin</code> , <code>stdout</code> , <code>stderr</code> anwendbar) String , der einen gültigen Pfad enthält
<code>FilterInputStream</code>	InputStream
<code>FilterOutputStream</code>	OutputStream
<code>FilterReader</code>	Reader
<code>InputStreamReader</code>	InputStream

LineNumberReader	Reader
ObjectInputStream	InputStream
ObjectOutputStream	OutputStream
OutputStreamWriter	OutputStream
PipedInputStream	PipedOutputStream Der parameterlose Vorgabekonstruktor erzeugt einen unverbundenen Strom.
PipedOutputStream	PipedInputStream Der parameterlose Vorgabekonstruktor erzeugt einen unverbundenen Strom.
PipedReader	PipedWriter
PipedWriter	PipedReader
PrintWriter	OutputStream Writer
PrintStream	OutputStream
PushbackInputStream	InputStream
PushbackReader	Reader
SequenceInputStream	InputStream Oder Inhalt eines Objekts dessen Klasse die Enumeration -Schnittstelle implementiert.
StringReader	String

Der [LineNumberReader](#) liefert ein Beispiel eines Stroms, der auf Basis eines anderen Stroms definiert wird. Im Falle des folgenden Beispiels wird ein LineNumberReader ausgehend von einem bestehenden FileReader erzeugt.

Wie bereits im [zweiten Kapitel angesprochen](#) unterstützt Java den Unicode-Zeichensatz. Durch ihn wird die plattformübergreifende Darstellung verschiedenster Zeichensätze ermöglicht. Als Erweiterung des klassischen ISO 8859-Teil 1 Zeichensatzes benötigt er jedoch generell 16 Bit zur Darstellung eines Symbols. Zusätzlich ist zu einem Unicode codierten Datenstrom die Codierungsschema, identifiziert durch ein eindeutiges Kürzel, anzugeben um die korrekte Darstellung zu ermöglichen.

Hierzu erlauben die Stream-Klassen [InputStreamReader](#) und [OutputStreamWriter](#) die explizite Spezifikation der Eingabe- bzw. Ausgabe Encodierung.

Jede Java-Implementierung muß mindestens folgende Code-Formate unterstützen: US-ASCII, ISO-8859-1, UTF-16BE (16-Bit Unicode im big-endian Format), UTF-16LE (dergleichen als little-endian) und UTF-16 (allgemeines 16-Bit Unicodeformat, *byte order mark* am Beginn des Stroms definiert verwendetes Anordnungsschema).

```

(1)import java.io.FileDescriptor;
(2)import java.io.FileOutputStream;
(3)import java.io.FileReader;
(4)import java.io.IOException;
(5)import java.io.LineNumberReader;
(6)import java.io.OutputStreamWriter;
(7)
(8)public class UnicodeWriter {
(9)    public static void main(String[] args) {
(10)        LineNumberReader lnr = null;
(11)        OutputStreamWriter osw = null;
(12)
(13)        String encoding, text;
(14)
(15)        try {
(16)            System.out.print("specify encoding:");
(17)            lnr = new LineNumberReader(new FileReader(FileDescriptor.in));
(18)            encoding = lnr.readLine();
(19)
(20)            System.out.print("Specify text to encode:");
(21)            text = lnr.readLine();
(22)
(23)            System.out.print("encoded text:");
(24)
(25)            osw = new OutputStreamWriter((new FileOutputStream(FileDescriptor.out)),
encoding);
(26)            osw.write(text);
(27)        } catch (IOException ioe) {
(28)            System.out.println("an IOException occurred\n"+ioe.getMessage() );
(29)        } finally {
(30)            try {
(31)                lnr.close();
(32)                osw.close();
(33)            } catch (Exception e) {
(34)                //ignore it
(35)            } //catch
(36)        } //finally
(37)    } //main()

```



```
(38)} //class UnicodeWriter
```

Beispiel 58: Schreiben in frei wählbarem Ausgabeencoding [UnicodeWriter.java](#)

Das Programm liest zunächst von der Standardeingabe die gewünschte Encodingdefinition und den auszugebenen Text als Zeichenkette.

Dann wird ein Ausgabestrom auf die Standardausgabe erzeugt, der das zuvor spezifizierte Encoding verwendet.

Unterstützt die Java-Implementierung das angegebene Codierungsschema nicht, schlägt die Erzeugung des Ausgabestroms fehl, und es wird ein Ausnahmeereignis erzeugt.

Zum Abschluß wird der Text unter Anwendung des definierten Encodingschemas über den Ausgabestrom ausgegeben.

Beispielinteraktionen:

```
specify encoding:US-ASCII
Specify text to encode:abcäöüß
encoded text:abc????
```

```
specify encoding:UTF8
Specify text to encode:ää
encoded text:a0C×
```

```
specify encoding:UTF-16LE
Specify text to encode:test
encoded text:t e s t
```

```
(1)import java.io.IOException;
(2)import java.io.FileDescriptor;
(3)import java.io.LineNumberReader;
(4)import java.io.FileReader;
(5)import java.io.FileWriter;
(6)
(7)
(8)public class Type {
(9)    public static void main(String[] args) {
(10)        FileReader fr = null;
(11)        FileWriter fw = null;
(12)        LineNumberReader lnr = null;
(13)        String line;
(14)
(15)        try {
(16)            fr = new FileReader(args[0]);
(17)            lnr = new LineNumberReader (fr);
(18)            fw = new FileWriter(FileDescriptor.out);
(19)
(20)            while ( (line = lnr.readLine()) != null) {
(21)                fw.write( lnr.getLineNumber()+": "+line+"\n" );
(22)            } //while
(23)        } catch (IOException ioe) {
(24)            System.out.println("an IOException occurred");
(25)            System.out.println( ioe.getMessage() );
(26)        } finally {
(27)            try {
(28)                fr.close();
(29)                fw.close();
(30)            } catch (IOException ioe) {
(31)                //ignore it
(32)            } //catch
(33)        } //finally
(34)    } //main()
(35)} //class Type
```

Beispiel 59: Zeilenweise Ausgabe einer Datei incl. Zeilennummern [Type.java](#)

Serialisierung von Objekten

Durch den Stromtyp `ObjectOutputStream` können vollständige Objekte geschrieben, und durch `ObjectInputStream` wieder in den Speicher eingeladen werden.

```

(1)import java.io.FileInputStream;
(2)import java.io.FileOutputStream;
(3)import java.io.IOException;
(4)import java.io.ObjectInputStream;
(5)import java.io.ObjectOutputStream;
(6)import java.io.Serializable;
(7)import java.util.Calendar;
(8)import java.util.GregorianCalendar;
(9)
(10)public class SerializeData {
(11)    public static void main(String[] args) {
(12)        //just for determining the current year
(13)        GregorianCalendar gregCal = new GregorianCalendar();
(14)
(15)        Person hans = new Person();
(16)        hans.name = new String("hans");
(17)        hans.yearOfBirth = 1950;
(18)        hans.age = gregCal.get(Calendar.YEAR) - hans.yearOfBirth;
(19)
(20)        System.out.println("object before serialization:\n" + hans.toString() );
(21)
(22)        ObjectOutputStream oos = null;
(23)
(24)        try {
(25)            oos = new ObjectOutputStream( new FileOutputStream("hans") );
(26)            oos.writeObject ( hans );
(27)        } catch (IOException ioe) {
(28)            System.out.println("an IOException occurred");
(29)            System.out.println( ioe.getMessage() );
(30)        } finally {
(31)            try {
(32)                oos.close();
(33)            } catch (IOException ioe) {
(34)                //ignore it
(35)            } //catch
(36)        } //finally
(37)
(38)        gregCal = null;
(39)        hans = null;
(40)
(41)        ObjectInputStream ois = null;
(42)        try {
(43)            Person anotherOne;
(44)            ois = new ObjectInputStream( new FileInputStream("hans") );
(45)            anotherOne = (Person) ois.readObject();
(46)
(47)            System.out.println( "object retrieved from file:\n"+ anotherOne.toString
(48)() );
(49)        } catch (IOException ioe) {
(50)            System.out.println("an IOException occurred while reading back object");
(51)        } catch (ClassNotFoundException cnfe) {
(52)            System.out.println("could not find class Person");
(53)        } finally {
(54)            try {
(55)                ois.close();
(56)            } catch (IOException ioe) {
(57)                //ignore it
(58)            } //catch
(59)        } //finally
(60)} //main()
(61)} //class SerializeData
(62)class Person implements Serializable {
(63)    public int yearOfBirth;
(64)    public String name;
(65)    transient int age;
(66)
(67)    public void finalize() {
(68)        System.out.println("object destroyed");
(69)    } //finalize()
(70)
(71)    public String toString() {
(72)        return("name="+name+"\n"+"yearOfBirth="+yearOfBirth+"\n"+"age="+age);
(73)    } //toString()
(74)} //class Person

```

Beispiel 60: Serialisieren und Laden eines Objekts [SerializeData.java](#)

BildschirmAusgabe:

```

object before serialization:
name=hans
yearOfBirth=1950
age=50
object retrieved from file:
name=hans
yearOfBirth=1950
age=0

```

Das [transiente Attribut](#) wird nicht in die Datei übernommen. Die Inhalte aller anderen Attribute werden gesichert, und können rückgelesen werden.

Beim (Wieder-)Einlesen eines Objektes wird versucht dessen Klassendefinition aus der entsprechenden Klassendatei zu laden. Ist dies nicht möglich, so wird ein Ausnahmeereignis vom Typ [ClassNotFoundException](#) generiert.

Anmerkung: Der Java-Serialisierungsmechanismus verhindert die mehrfache Ablage desselben Objektes im Bytestrom.

Für die häufig umzusetzende Aufgabe der Eingabeformatprüfung, und anschließenden Klassifizierung in bestimmte Kategorien steht die Klasse [StreamTokenizer](#) zur Verfügung.

```

(1)import java.io.StreamTokenizer;
(2)import java.io.FileReader;
(3)import java.io.FileDescriptor;
(4)import java.io.IOException;
(5)
(6)public class TokenTest {
(7)    public static void main(String[] args) {
(8)        StreamTokenizer st = null;
(9)
(10)        int    op1=0,
(11)              op2=0;
(12)        try {
(13)            st = new StreamTokenizer(new FileReader(FileDescriptor.in));
(14)
(15)            while(st.nextToken() == StreamTokenizer.TT_NUMBER)
(16)                op1 = (op1*10) +(int) st.nval;
(17)
(18)            while(st.nextToken() == StreamTokenizer.TT_NUMBER)
(19)                op2 = (op2*10) + (int) st.nval;
(20)
(21)            System.out.println(op1+" "+op2+"="+ (op1+op2));
(22)        } catch (IOException ioe) {
(23)            System.out.println("an IOException occurred\n"+ioe.getMessage() );
(24)        } //catch
(25)    } //main()
(26)} //class TokenTest

```

Beispiel 61: Einfache Addition zweier Zahlen unter Verwendung des StringTokenizers [TokenTest.java](#)

Die beiden Operanden können durch ein beliebiges nicht numerisches Zeichen voneinander abgetrennt werden, ebenso kann das Berechnungsende erklärt werden.

Zum Zugriff auf **(G-)ZIP komprimierte Dateien** bieten die Klassen [ZipInputStream](#) und [GZIPInputStream](#) bzw. [ZipOutputStream](#) und [GZIPOutputStream](#) Lese- bzw. Schreibströme an.

▲ 3.2.2 Threads und Nebenläufigkeit

Java bietet mit den sog. Programmfäden engl. *Threads* die Möglichkeit an, parallele leichtgewichtige Prozesse direkt in der Hochsprache zu definieren und zu kontrollieren. Hierbei werden keine Anforderungen an eine spätere Unterstützung dieses Konzepts durch die tatsächliche physische Hardware gestellt; der gesamte Mechanismus ist rein Hochsprachen-basiert.

Als Bestandteil des [automatisch importierten](#) Paketes `java.lang` stehen Threads in jeder Applikation und jedem Applet ohne zusätzliche Aufwende zur Verfügung.

Weiterführende Informationen: [Java Language Specification, chap. 17](#) und [Java JVM Spezifikation, chap. 8](#).

Inhaltlich unterscheidet sich ein Thread nur in marginalen Modifikationen von einer gewöhnlichen Klasse. Hauptunterschied ist die eigenständige aktive und nebenläufige Ausführung von Objekte einer solchen Klasse.

Voraussetzungen zur Erzeugung eines Threads:

- Die Klasse spezialisiert die Standard-API-Klasse [Thread](#).
Wie alle Klassen, die nebenläufig ausgeführt werden sollen implementiert auch [Thread](#) die Schnittstelle [Runnable](#).

Sie definiert die Operation `run()`.

- Die Klasse überschreibt die Methode `run()`.

Sie wird explizit durch die Methode `start()` aufgerufen. `start()` ist asynchron und kehrt sofort nach Erzeugung des Threads zum Aufrufer zurück.

Hinweis: Die Methode `run()` sollte nicht direkt aufgerufen werden! Bei der direkten Ausführung unterbliebe die notwendige Initialisierung; insbesondere wäre der dann „gewöhnliche“ Methodenaufruf nicht asynchron, und das neue Objekt würde nicht nebenläufig ausgeführt.

```
(1)public class Threads1 {
(2)    public static void main(String[] args) {
(3)        HelloThread northGerman = new HelloThread( "Moin Moin" );
(4)        HelloThread southGerman = new HelloThread( "Gruess Gott" );
(5)
(6)        northGerman.start();
(7)        southGerman.start();
(8)    } //main()
(9)} //class Threads
(10)
(11)class HelloThread extends Thread {
(12)    protected String greetingText;
(13)
(14)    public HelloThread (String greetingText) {
(15)        this.greetingText = greetingText;
(16)    } //constructor
(17)
(18)    public void run() {
(19)        while (true) {
(20)            try {
(21)                Thread.sleep(500);
(22)            } catch (InterruptedException ie) {
(23)                System.out.println("an InterruptedException occurred\n"+ie.toString
(24)                )+"\\n"+ie.getMessage() );
(25)            } //catch
(26)            System.out.println( greetingText);
(27)        } //while
(28)} //class HelloThread
```

Beispiel 62: Zwei einfache Threads [Threads1.java](#)

Das Programm gibt die beiden Texte *Moin Moin* und *Gruess Gott* jeweils im Wechsel durch separate Threads aus. Besonders fällt auf, daß die Applikation nach Abarbeiten der letzten Anweisung der `main`-Methode nicht terminiert.

Java-Programme die (noch) *Vordergrund*-Threads ausführen terminieren nicht am Ende der `main`-Methode, sondern führen weiterhin die noch laufenden Threads -- bis zu deren eigenständigem Terminieren oder Abbruch -- aus. Die zweite Threadklasse bilden die [Daemon-Threads](#). Verfügt ein laufendes Programm ausschließlich über solche Hintergrund-Threads terminiert es am Ende der `main`-Methode wie gewohnt. Demnach läßt sich der bisher bekannte Programmtyp als nebenläufige Applikation mit dem Vordergrundthread `main` betrachten. Terminiert dieser Thread, so terminiert auch die gesamte Applikation.

Hinweis: Programme mit laufenden Vordergrundthreads lassen sich durch Beenden der virtuellen Maschine mittels [System.exit\(int\)](#) terminieren.

Das Beispiel enthält auch bereits zwei der möglichen Status innerhalb des Lebenszyklus eines Threads, nämlich *erzeugt* (aber noch nicht in Ausführung befindlich, nach Aufruf des Konstruktors) und *in Ausführung* nach dem Starten durch den Aufruf der Methode `run()`.

Die möglichen Threadzustände können jedoch noch durch weitere Methoden beeinflusst werden:

- [start\(\)](#)
Erzeugung eines Threads (starten) aus einem bereits erstellten Thread-Objekts.
- [sleep\(long\)](#) und [sleep\(long, int\)](#)
Anhalten eines Threads für eine durch die Übergabeparameter festgelegte Zeitspanne. Während der Ausführungspause bleiben gesetzte Sperren (Monitore) erhalten.
- [join\(\)](#), [join\(long\)](#) und [join\(long, int\)](#)
Wartet auf das Terminieren des erzeugten Threads. (Quasi synchrone Variante von `run()`). Optional kann eine maximale abzuwartende Zeitspanne definiert werden.
- [yield](#)
Freiwillige Abgabe der Zeitscheibe.
- [destroy\(\)](#)
Zerstört Thread ohne vorherige Aufräumarbeiten; insbesondere werden gesetzte Sperren nicht freigegeben.

Hinweis:

Die beiden [stop](#)-Methoden, sowie die Methoden [suspend](#) und [resume](#) sind seit der Version 1.3 als *deprecated* gekennzeichnet und sollten nicht mehr verwendet werden!

Der Grund liegt in der, mit der durch die API zugesicherten, sofortigen Unterbrechung. Dabei kann die Unterbrechung auch innerhalb eines kritischen Abschnittes erfolgen, wodurch es zu verschiedensten Inkonsistenzen und weiteren Folgeproblemen kommen kann.

Hinzu kommen von [java.lang.Object](#) ererbte Methoden:

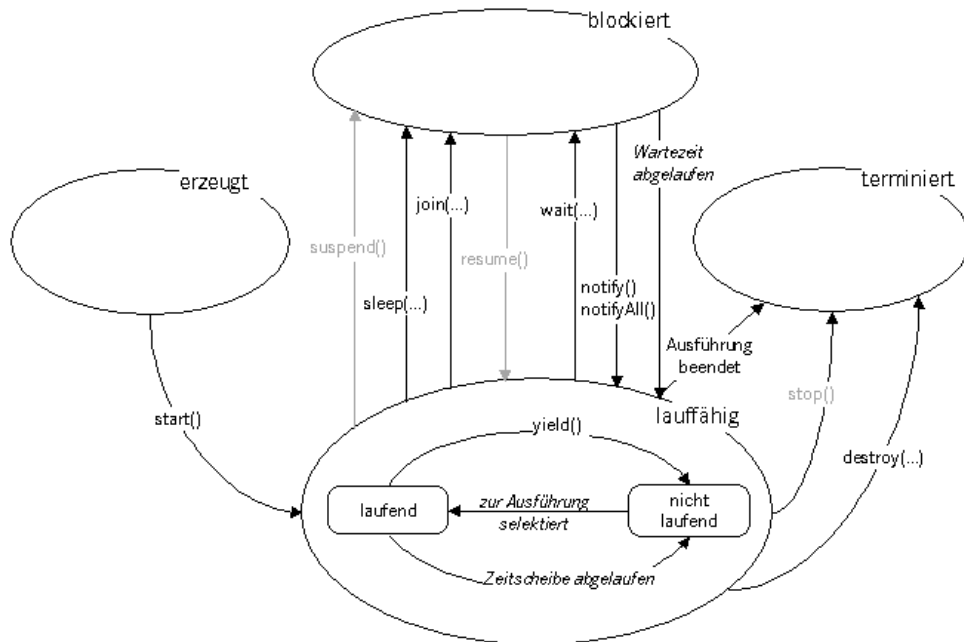
- [wait\(\)](#)
[wait\(long\)](#)
[wait\(long, int\)](#)

Hält Thread beim Zugriff auf ein bestimmtes Objekt an, bis ein anderer Thread die Methode [notify\(\)](#) oder [notifyAll\(\)](#) ausführt.

Zur Synchronisation werden Monitore verwendet.

- [notify\(\)](#)
[notifyAll\(\)](#)

Führt mit der Ausführung eines im Threads fort, der beim Zugriff auf ein gesperrtes Objekt in den Wartezustand versetzt wurde.



Die Graphik zeigt die verschiedenen möglichen Zustände eines Threads. Als *deprecated* gekennzeichnete Methoden sind grau unterlegt.

Das interne Scheduling aller laufenden Threads erfolgt prioritätsgesteuert. Die **Prioritäten** zugreifbarer (d.h. eigen erzeugter) Threads kann erfragt und gezielt verändert werden.

Die Threadpriorität wird als ganzzahliger Wert angegeben. Plattformspezifisch kann dieser Wert variieren; jedoch kann die konkrete Unter- und Obergrenze über die Konstanten [MIN_PRIORITY](#) bzw. [MAX_PRIORITY](#) zur Laufzeit erfragt werden. Ohne manuellen Eingriff werden neue Threads mit der durch [NORM_PRIORITY](#) definierten Priorität erzeugt.

Methoden zur Beeinflussung der Thread-Priorisierung:

- [int getPriority\(\)](#)
liefert die Priorität des Threads.
- [setPriority\(int\)](#)
versucht die Priorität auf den übergebenen Wert zu setzen. Dies kann wegen mangelnder Berechtigung, oder ungültiger Übergabeparameter, fehlschlagen. In diesem Falle wird ein Ausnahmeereignis der Klasse [SecurityException](#), bzw. -- bei ungültigen Übergabeparametern -- [IllegalArgumentException](#) erzeugt.

Aus Gründen der vereinfachten Verwaltung können Threads in Gruppen (API-Klasse [ThreadGroup](#)) zusammengefaßt und damit gebündelt beeinflußt werden.

Operationen zur Zustandsänderung die auf einzelnen Threads wirken, können auch auf **Thread-Gruppen** angewendet werden.

Die Verwaltung von solchen Thread-Bündeln ist in der API-Klasse [ThreadGroup](#) zusammengefaßt; die Gruppenzugehörigkeit eines spezifischen Threads kann durch [getThreadGroup\(\)](#) ermittelt werden.

Daemon-Threads:

wie bereits angerissen gibt es neben den „normalen“ Anwenderthreads auch die Klasse der Daemon-Threads. Ihr Hauptunterscheidungsmerkmal zu den Anwenderthreads ist das Charakteristikum, daß die JVM auch terminiert, wenn sich Threads dieser Kategorie in Ausführung befinden. Dies prädestiniert sie für Hintergrundaufgaben wie Verwaltungs- oder Überwachungstätigkeiten. Der Typ eines Threads kann vor dessen Ausführungsbeginn durch den `start()`-Aufruf mittels der Methode [setDaemon\(boolean\)](#) festgelegt, und zur Laufzeit mit [isDaemon\(\)](#) jederzeit während der Ausführung erfragt, werden.

```

(1)public class IsPrime {
(2)    public static void main(String[] args) {
(3)        PrimitivePrimeTest ppt;
(4)        StatusInformation myInfo = new StatusInformation();
(5)        myInfo.setDaemon(true); //declare as daemon thread
(6)        myInfo.setPriority( (Thread.NORM_PRIORITY + 2) <= Thread.MAX_PRIORITY ? Thread.
NORM_PRIORITY + 2 : Thread.MAX_PRIORITY );
(7)        myInfo.start();
(8)
(9)        ThreadGroup workerThreads      = new ThreadGroup("worker threads");
(10)
(11)        for (int i = Integer.parseInt(args[0]); i <= Integer.parseInt(args[1]); i++) {
(12)            ppt = new PrimitivePrimeTest(workerThreads, i, "calculating " +i );
(13)            ppt.start();
(14)        } //for
(15)
(16)    } //main
(17)} //class IsPrime2
(18)
(19)class PrimitivePrimeTest extends Thread {
(20)    protected int numberToTest;
(21)    boolean earlyExit=false;
(22)
(23)    public PrimitivePrimeTest(ThreadGroup tg, int numberToTest, String threadName) {
(24)        super (tg, threadName); //call to super's class constructor
(25)        this.numberToTest = numberToTest;
(26)    } //constructor
(27)
(28)    public void run() {
(29)        if (numberToTest == 1)
(30)            earlyExit = true;
(31)        for (int i=2; i < ((int) Math.sqrt(numberToTest))+1; i++) {
(32)            try {
(33)                Thread.sleep(1000);
(34)            } catch (InterruptedException ie) {
(35)                //ignore it
(36)            } //catch
(37)            if (numberToTest % i == 0) {
(38)                earlyExit=true;
(39)                break;
(40)            } //endif
(41)        } //for
(42)        if (earlyExit != true)
(43)            System.out.println(numberToTest+" is prime");
(44)    } //run()
(45)} //class PrimitivePrimeTest
(46)
(47)class StatusInformation extends Thread {
(48)    public void run() {
(49)        while (true) {
(50)            System.out.println("no of currently running threads: "+Thread.activeCount
() );
(51)            try {
(52)                Thread.sleep(500);
(53)            } catch (InterruptedException ie) {
(54)                //ignore it
(55)            } //catch
(56)        } //while
(57)    } //run()
(58)} //class StatusInformation

```

Beispiel 63: Ein einfacher Primzahlenprüfer [IsPrime.java](#)

Beispielablauf:

```

$java IsPrime 1 25
no of currently running threads: 2
2 is prime
3 is prime
no of currently running threads: 24
no of currently running threads: 24
5 is prime
7 is prime
no of currently running threads: 11
no of currently running threads: 11
11 is prime
13 is prime
no of currently running threads: 6
no of currently running threads: 6
17 is prime

```

```

19 is prime
23 is prime
no of currently running threads: 3
no of currently running threads: 3

```

Das Programm prüft in jeweils einem eigenen Thread, ob die Ganzzahlen im Intervall zwischen den gegebenen Grenzen prim sind. Entdeckte Primzahlen werden mit einer entsprechenden Meldung ausgegeben.

Um die unterschiedliche Ausführungsdauer der jeweiligen Threads hervorzuheben führt jeder Programmfaden nur eine Berechnung pro Sekunde aus.

Die Berechnungsthreads sind alle in einer eigenen Threadgruppe (workerThreads) zusammengefaßt. Jeder Thread innerhalb dieser Gruppe wird durch die Zeichenkette *calculating* gefolgt von der zu prüfenden Zahl benannt.

Zusätzlich gibt ein Daemon-Thread alle halbe Sekunde die Anzahl der (noch) aktiven Threads aus. Dieser Thread wird automatisch durch die JVM nach dem Abarbeiten des letzten aktiven Anwenderthreads terminiert. Die Priorität des Daemon-Threads ist um zwei gegenüber dem Vorgabewert erhöht, sofern dadurch nicht die maximal erlaubte Priorisierung überschritten wird.

Synchronisation von Methodenzugriffen:

```

(1)import java.io.DataOutputStream;
(2)import java.io.FileOutputStream;
(3)import java.io.FileInputStream;
(4)import java.io.DataInputStream;
(5)import java.io.IOException;
(6)import java.io.FileNotFoundException;
(7)
(8)public class Threads2 {
(9)    public static void main(String[] args) {
(10)        for (int threadCount=0; threadCount < Integer.parseInt(args[0]); threadCount++)
(11)            (new IncrementCounter()).start();
(12)    } //main()
(13)} //class Threads2
(14)
(15)class IncrementCounter extends Thread {
(16)    public void run() {
(17)        int counterValue;
(18)        DataInputStream dis = null;
(19)        DataOutputStream dos = null;
(20)
(21)        try {
(22)            for (int i=0; i<10; i++) {
(23)                dis = new DataInputStream( new FileInputStream( "counter" ) );
(24)                counterValue = dis.readInt();
(25)                dis.close();
(26)
(27)                System.out.println(counterValue+" read by thread "+ (Thread.
currentThread().getName() ));
(28)
(29)                dos = new DataOutputStream( new FileOutputStream( "counter",
false ) );
(30)                dos.writeInt( ++counterValue );
(31)                dos.close();
(32)            } //for
(33)        } catch (FileNotFoundException fnfe) {
(34)            System.out.println("file counter could not be opened!\n"+fnfe.toString()
+"\n"+fnfe.getMessage() );
(35)            System.exit(1);
(36)        } catch (IOException ioe) {
(37)            System.out.println("an IOException occurred in thread "+(Thread.
currentThread().getName()+"!\n"+ioe.toString()+"\n"+ioe.getMessage() );
(38)            System.exit(1);
(39)        } //catch
(40)    } //run()
(41)} //class IncrementCounter

```

Beispiel 64: Gemeinsames Inkrementieren eines Zählers in einer Datei [Threads2.java](#)

Hilfsprogramme: [manuelles Rücksetzen des Zählerstandes auf 0](#), [manuelles Auslesen des Zählerstandes und Ausgabe auf Standardausgabe](#)

Das Programm liest den Stand einer ganzzahligen Variable aus einer Datei aus, erhöht um Eins und schreibt ihn zurück in dieselbe Datei, unter Verlust des alten Standes (Überschreiben). Die beschriebene Vorgangsfolge wird im Multithreading-Betrieb durch mehrere Ausführungsfäden nebenläufig durchgeführt.

Beim Auffangen eines Ausnahmeereignisses wird die virtuelle Maschine terminiert, da das einzelne Terminieren nur eines Threads das Ergebnis verfälschen würde.

Tritt zwischen dem Auslesevorgang aus der Datei, und dem Rückschreiben des modifizierten Wertes ein Threadwechsel ein, so kommt es zum Phänomen des *lost updates*.

mögliche Bildschirmausgaben:

Nach dem Einlesen der Zahl 5 durch Thread-33 tritt, vor ihrem inkrementierten Rückschreiben, der Threadwechsel ein, wodurch der nachfolgend ausgeführte Thread-34 dieselbe Zahl nochmals ausliest.

```
$java Threads2
...
3 read by thread Thread-28
4 read by thread Thread-30
5 read by thread Thread-33
5 read by thread Thread-34
6 read by thread Thread-42
7 read by thread Thread-45
...
```

Sicherlich ein Extremum dieses Verhaltens stellt das nachfolgende Beispiel dar:

mögliche Bildschirmausgaben:

Threadwechsel tritt jeweils direkt nach dem Einlesen auf. Als Konsequenz wird derselbe Zahlenwert mehrfach durch verschiedene Threads gelesen.

```
$java threads2
...
10 read by thread Thread-35
10 read by thread Thread-10
10 read by thread Thread-36
10 read by thread Thread-37
10 read by thread Thread-14
10 read by thread Thread-38
10 read by thread Thread-12
10 read by thread Thread-39
10 read by thread Thread-16
10 read by thread Thread-41
10 read by thread Thread-17
10 read by thread Thread-19
10 read by thread Thread-42
10 read by thread Thread-18
10 read by thread Thread-40
10 read by thread Thread-0
10 read by thread Thread-11
...
```

Um Daten-Inkonsistenzen zur Laufzeit zu verhindern, wie sie potentiell entstehen könnten, wenn zwei Programmfäden gleichzeitig dieselbe Methode ausführen oder zeitlich verschränkt dieselbe kritische Ressource benutzen, ist mit dem (aus Kapitel zwei bekannten) Schlüsselwort [synchronized](#) ein Hochsprachenmechanismus gegeben um mehrere Aufrufer gezielt zu serialisieren.

Technisch handelt es sich dabei um das aus den Betriebssystemen bekannte Konzept der Monitore (siehe [C. A. R. Hoare: Monitors: An Operating System Structuring Concept](#)) zur Synchronisation des Zugriffs auf kritische Abschnitte.

Jedoch besteht bei dieser Vorgehensweise die Gefahr eines **Deadlocks** durch wechselseitiges Blockieren!

Durch synchronisierte Blöcke können nicht nur einzelne Methoden, sondern Bündel von Zugriffen gemeinsam geschützt werden.

Hierzu wird dem Schlüsselwort `synchronized` ein auswertbarer Ausdruck nachgestellt, der die Resource bezüglich der zu synchronisieren ist bezeichnet.

Im Falle des betrachteten Beispiels ist daher hinsichtlich der gemeinsam beanspruchten (und daher kritischen) Ressource der Zählerstandsdatei zu synchronisieren.

```
(1)import java.io.DataInputStream;
(2)import java.io.DataOutputStream;
(3)import java.io.File;
(4)import java.io.FileInputStream;
(5)import java.io.FileNotFoundException;
(6)import java.io.FileOutputStream;
(7)import java.io.IOException;
(8)
(9)public class Threads21 {
(10)    public static synchronized void main(String[] args) {
(11)        File counterFile = new File ("counter");
(12)
(13)        for (int threadCount=0; threadCount < Integer.parseInt(args[0]); threadCount++)
(14)            (new IncrementCounter(counterFile)).start();
(15)    } //main()
(16)} //class Threads21
(17)
(18)class IncrementCounter extends Thread {
(19)    File    counterFile;
(20)    public IncrementCounter(File rwFile) {
(21)        counterFile = rwFile;
(22)    } //constructor
(23)
(24)    public void run() {
```



```

(25)         int counterValue;
(26)         DataInputStream dis = null;
(27)         DataOutputStream dos = null;
(28)
(29)         try {
(30)             for (int i=0; i<10; i++) {
(31)                 synchronized (counterFile) {
(32)                     dis = new DataInputStream( new FileInputStream
( counterFile ));
(33)                     counterValue = dis.readInt();
(34)                     dis.close();
(35)
(36)                     System.out.println(counterValue+" read by thread "+ (Thread.
currentThread()).getName() );
(37)
(38)                     dos = new DataOutputStream( new FileOutputStream
( counterFile ));
(39)                     dos.writeInt( ++counterValue );
(40)                     dos.close();
(41)                 } //synchronized
(42)             } //for
(43)         } catch (FileNotFoundException fnfe) {
(44)             System.out.println("file counter could not be opened!\n"+fnfe.toString()
+"\n"+fnfe.getMessage() );
(45)             System.exit(1);
(46)         } catch (IOException ioe) {
(47)             System.out.println("an IOException occurred in thread "+(Thread.
currentThread()).getName()+"!\n"+ioe.toString()+"\n"+ioe.getMessage() );
(48)             System.exit(1);
(49)         } //catch
(50)     } //run()
(51)} //class IncrementCounter

```

Beispiel 65: Lost-Update-freie Variante des vorhergehenden Beispiels [Threads21.java](#)

Auffallendstes Kennzeichen der Synchronisation mittels Schlüsselwort `synchronized` ist es, daß gemeinsam genutzte Objekte existieren müssen, um die alle Threads konkurrieren.

Auf den Einsatzfall einer eher losen Kopplung der einzelnen Ausführungseinheiten sind die Methoden `wait` und `notify` ausgelegt. Der wohl bekannteste Vertreter dieser Problemklasse ist das klassische *Erzeuger-Verbraucher-Schema*, in dem zwei grundlegend verschiedene Rollen, die des Erzeugers und die des zeitlich nachgelagerten -- und daher zu synchronisierenden -- Verbrauchers unterschieden werden.

Für diese Problemklasse bietet Java das Schlüsselwort `wait` an. Es erlaubt das Abwarten eines Ereignisses, welches durch `notify` angezeigt wird.

Das Standardschema zur Benutzung lautet:

```

synchronized doWhenCondition() {
    while (!condition)
        wait();
    ...Bedingung true...
} //synchronized

```

Als Randbedingung gilt die zwingende Einbettung in eine als `synchronized` deklarierte Methode, um die Modifikation der While-Bedingung durch nebenläufig ausgeführte Threads zu verhindern.

Der `wait`-Aufruf suspendiert die Ausführung des Prozesses, und gibt gleichzeitig (atomar) seine gesetzten Sperren frei.

Das Benutzungsschema für `notify` lautet:

```

synchronized changeCondition() {
    ...Änderung von Werten, die in der Bedingung auftreten...
    notify();
} //synchronized

```

Die `wait`-Methode steht in verschiedenen Ausgestaltungen zur Verfügung:

[wait\(\)](#) wartet bis zur Wiedererweckung durch `notify`; gleiche Wirkung wie `wait(0)`.

[wait\(long\)](#) wartet bis zum Ablauf des durch den Übergabeparameter fixierten Zeitraumes in Millisekunden auf den Aufruf von `notify()`.

[wait\(long, int\)](#) wartet bis zum Ablauf des durch die Übergabeparameter spezifizierten Zeitraumes -- der `long`-Wert wird als Millisekunden interpretiert, zu dem die als `int`-Wert gegebenen Nannosekunden addiert werden -- auf den Aufruf von `notify()`.

Zur Wiedererweckung wartender Threads kann die parameterlose Methode [notify\(\)](#) benutzt werden; sie erweckt maximal einen wartenden Thread.

Den Wiederanlauf beliebig vieler wartender Threads ermöglicht [notifyAll\(\)](#).

Anmerkungen:

- Konstruktoren sind implizit *synchronized*, da ein Objekt nur genau von einer aktiven Einheit gleichzeitig erzeugt werden kann.

- Synchronisierte Klassenmethoden ziehen eine Sperre auf Klassenebene nach sich.
- synchronized-Angaben können in ererbten Klassen überschrieben werden. Das Verhalten der Superklasse bleibt hiervon jedoch unberührt.
- Synchronisierte Abschnitte erfordern zusätzlichen Byte-Code zur Zugriffskontrolle und notwendigen Serialisierung, daher sinkt durch Nutzung dieses Mechanismus im Allgemeinen die Ausführungsgeschwindigkeit.

Ermitteln von Threadinformation:

Die Anzahl der laufenden aktiven Threads der aktuellen Gruppe kann durch `activeCount()` ermittelt werden. Informationen über jeden Thread der virtuellen Maschine liefern die Methoden `getName()` und `setName(String)`. Diese beiden Methoden erlauben die Abfrage, bzw. Definition eines Namens für jeden Thread. Bei der Erzeugung eines Threads wird automatisch ein Name durch die virtuelle Maschine vergeben. (SUNs JVM setzt hier die Zeichenkette Thread-*i*, wobei *i* die laufende Nummer des Threads ist.) `enumerate(Thread[])` befüllt den als Parameter übergebenen Array mit Verweisen auf die derzeit aktiven Thread Objekte. Der aktuelle Thread kann durch die statische Methode `currentThread()` ermittelt werden.

Außer durch Spezialisierung der Klasse Thread kann durch Implementierung des **Interfaces Runnable** Nebenläufigkeit erzeugt werden.

Die Schnittstelle definiert ausschließlich die Operation run. Auch die Umsetzung der Klasse Thread nutzt in den vordefinierten Konstruktoren dieses Interface. Durch den Methodenaufruf `start()` wird im zugehörigen nebenläufigen Objekt per `0.run()` (*0* ist hierbei vom Typ Runnable) die Ausführung begonnen.

Hinweis: Dieser Anwendungsfall ist insbesondere für Applets von Bedeutung, da hier keine Spezialisierung von Thread erfolgen kann, da zwingend von Klasse Applet abgeleitet werden muß.

([Beispiel dazu](#))

[Mehr zu Java Threads erfahren ... \(eigene Vorlesung zum Thema\)](#)

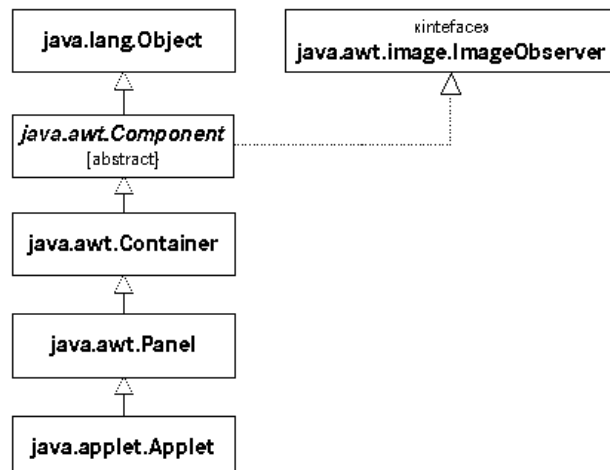
▲ 3.2.3 Applets

Applets sind Java-Programme die innerhalb einer Browserumgebung ausgeführt werden. Daher gelten für sie einige besondere Charakteristika, die sie von den bisher betrachteten Java-Applikationen unterscheiden:

- Ein Applet muß innerhalb einer (X)HTML-Seite referenziert werden, es kann nicht eigenständig ausgeführt werden.
- Die Applet-Hauptklasse muß zwingend von `Applet` abgeleitet sein.
- Ein Applet verfügt über keine main-Methode. Stattdessen erzeugt der Browser ein Objekt der Hauptklasse und führt die Methoden `init` und `start` automatisch aus.
- Durch die Browereinbettung, konkret: dessen **Sicherheitskonzept**, ist es einem Applet nicht gestattet auf die lokalen Ressourcen (Dateisystem, etc.) des Rechners zuzugreifen. Als Ausnahme hierzu können spezielle „vertrauenswürdige“ Applets erstellt werden, die durch den Programmierer **signiert** sein müssen.
- Applets bieten keine Textschnittstelle wie (Kommandozeilen-)Applikationen. Daher muß zwingend eine graphische Benutzeroberfläche erstellt werden.

Hinweis: Zu Testzwecken kann ein Applet auch durch einen Applet-Viewer, wie der u.a. [in SUNs JDK enthalten](#) ist, ausgeführt werden. Hierfür wird neben dem Applet selbst eine minimale HTML-Seite benötigt.

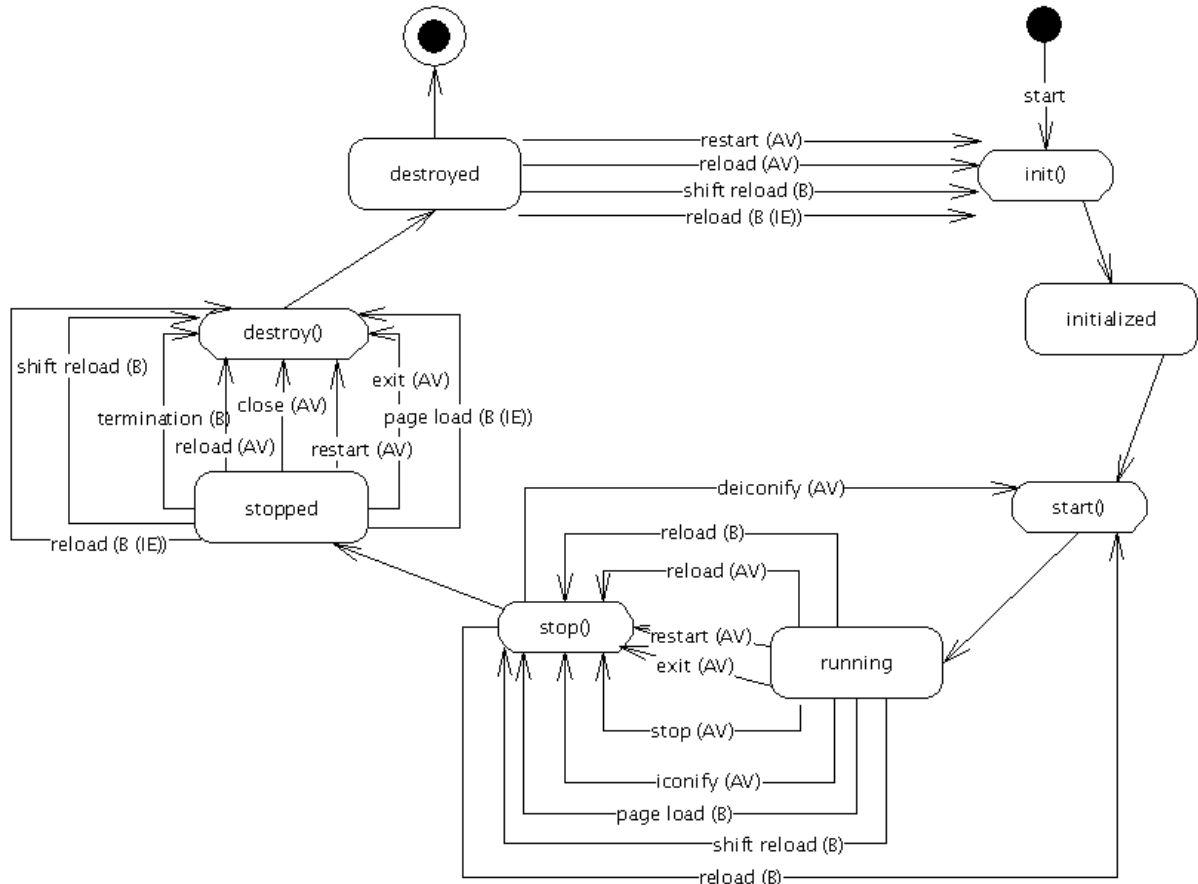
Die Klassenhierarchie von Applet und ihre Klassen:



- Panel: einfachste Container-Klasse. Sie stellt Platz zum Einhängen beliebiger weiterer Komponenten zur Verfügung.
- Container: Generischer Container des *Abstract Window Toolkit* (AWT), der andere AWT-Komponenten enthalten kann.
- Component: Objekt das über eine graphische Repräsentation verfügt, die am Bildschirm angezeigt werden kann, und die die Möglichkeit zur Benutzerinteraktion bietet. Beispiele: Schaltflächen (Button), Check Boxen und Scrollbars.

Lebenszyklus eines Applets -- die **zentralen Methoden der Klasse Applet**:

- [init\(\)](#)
Durch den Browser nach dem Laden des Applets, jedoch noch vor Beginn seiner Ausführung, aufgerufen.
Verwendung: Parameter einlesen, Laden von Bildern und Audiodateien, Objekterzeugung.
- [start\(\)](#)
Direkt nach Abschluß der `init()`-Methode durch den Browser aktiviert. Darüberhinaus wird diese Methode immer dann aufgerufen, wenn die beherbergende Web-Seite neu geladen wird.
Verwendung: Starten von Animationen, Audiowiedergabe.
- [stop\(\)](#)
Durch den Browser aufgerufen, wenn die beherbergende Seite nicht mehr angezeigt wird.
Verwendung: Beenden von Multimediawiedergaben.
- [destroy\(\)](#)
Im Anschluß an `stop()` durch den Browser aufgerufen, um die durch das Applet belegten Systemressourcen zurückzufordern.
Verwendung: Explizite Speicher-Freigabe (z.B. [flush](#)-Methode für Images).



Das Zustandsübergangsdiagramm zeigt die verschiedenen Status im Leben eines Applets.

Die Übergänge sind mit den auslösenden Ereignissen beschriftet. Ist kein Ereignis angetragen, so erfolgt der Übergang automatisch ohne weitere Bedingungen. In Klammern ist die jeweils ereignisauslösende Umgebung (AV für Appletviewer und B für Web-Browser) angegeben. Im Allgemeinen verhalten sich die beiden großen Browser gleich. Weicht das Verhalten eines Browsers ab, so ist dies explizit angegeben (IE kennzeichnet hierbei den MS Internet Explorer).

Weitere wichtige Methoden, die von den Superklassen von Applet geerbt werden sind:

- [paint\(Graphics\)](#)
Wird zum vollständigen Neuzeichnen des Applets aufgerufen.
- [update\(Graphics\)](#)
Aktualisiert Fensterinhalt.

Der Aufruf der **Methode `repaint()`** erzeugt einen Aufruf an die `update`-Methode der Komponente.

`repaint()` stellt die asynchrone Variante des direkten Aufrufes von `paint` mit dem aktuellen `Graphics`-Objekt (`paint ( getGraphics() )`) dar. Die Methode garantiert das Neuzeichnen nicht, im Falle hoher Auslastung der virtuellen Maschine kann es auch unterbleiben.

Die **benötigten (X)HTML-Strukturen**:

Leider existieren derzeit verschiedene Varianten ein Java-Applet in eine Hypertextseite einzubinden.

Zunächst die „klassische“ Applet-Referenz: `<applet code="className" width="Breite in Pixel" height="Höhe in Pixel">`

Zusätzlich können (gemäß HTML v4.01 Standard) noch weitere Attribute (Namen-Wert-Paare wie "code" oder "width") definiert werden:

`codebase` -- ermöglicht die relative Interpretation von Pfadnamen ausgehend vom in `codebase` definierten Katalog

`archive` -- ZIP komprimiertes Archiv das Applet gesucht wird.

`alt` -- zusätzlicher erläuternder Text.

name -- eindeutige Benennung des Applets innerhalb der beherbergenden Seite.
 align -- horizontale Ausrichtung auf der Seite.
 hspace -- horizontaler Abstand des Applets zur Umgebung.
 vspace -- vertikaler Abstand zur Umgebung.

Ferner existieren noch einige proprietäre Attribute, die jedoch nicht durch alle Browser unterstützt werden.

Parameterübergabe an Applets:

erfolgt durch Name-Wert-Paare der Form `<param name="..." value="...">`, die in den Grenzen der Applet-Tags eingeschlossen sind.

```
<applet code="HelloWorldApplet.class" width="100" height="100">
  <param name="color" value="green">
</applet>
```

Mit XHTML v1.0, dem Nachfolger des HTML v4.01 Standards, wurde die Applet-Syntax für veraltet (*deprecated*) erklärt, und nunmer ausschließlich die Object-Variante zugelassen. Über die Änderung der Tag-Namen hinausgehend dürfte die Plazierung der Referenz auf die Java-Klassendatei, und die explizite Typangabe, als Parameter die augenfälligste Änderung sein. Diese Syntaxvariante wird jedoch, obgleich Standard, noch nicht von allen derzeit gängigen Browservarianten unterstützt.

```
<object>
  <param name="type" value="application/x-java-applet">
  <param name="JAVA_CODE" value="HelloWorldApplet.class">
```

Hinweis: In älteren Referenzen findet sich teilweise noch die ursprüngliche HotJava-Syntax, welche ein `app`-Tag definiert. Ferner enthält dieses die Referenz auf die Klasse als Attribut, jedoch ohne die Dateiextension `.class` explizit anzuführen.

Das Hello World-Applet:

Das absolute Minimalapplet, es schreibt lediglich den bekannten Schriftzug, kann folgendermaßen umgesetzt werden:

```
(1)import java.applet.Applet;
(2)import java.awt.Graphics;
(3)
(4)public class HelloWorldApplet extends Applet {
(5)
(6)    public void paint(Graphics g) {
(7)        g.drawString(    getParameter("displayText"),
(8)                                Integer.parseInt(getParameter("xPos")),
(9)                                Integer.parseInt(getParameter("yPos"))) );
(10)    } //paint()
(11)    public String getAppletInfo() {
(12)        return( "written by Mario Jeckle\nversion: 1.1\nCopyright (c) by Mario Jeckle");
(13)    } //getAppletInfo()
(14)    public String[][] getParameterInfo() {
(15)        String appInfo[][] = {
(16)            {"displayText", "anyString", "string to display"},
(17)            {"xPos", "numeric value", "horizontal position"},
(18)            {"yPos", "numeric value", "vertical position"}
(19)        };
(20)        return appInfo;
(21)    } //getParameterInfo()
(22)} //end class HelloWorldApplet
```

Beispiel 66: Hello World als Java-Applet [HelloWorldApplet.java](#)

[Applet ausführen](#)

Interaktion mit der Ausführungsumgebung

Durch verschiedene vordefinierte Methoden und Schnittstellen kann jedes Applet mit seiner Ausführungsumgebung, dem Browser oder Applet Viewer, in Interaktion treten. Hierdurch können von Seiten des Applets auch Dienste wie Caching, Anzeigen einer Nachricht, Abspielen von Audiodateien der Ausführungsumgebung benutzt werden.

Einfachster Fall ist die Methode [getAppletInfo](#). Sie liefert eine Zeichenkette zurück, welche üblicherweise Informationen über den Autor, die Version und die Rechte am Applet beinhaltet.

Eine ähnliche Funktion erfüllt die Methode [getParameterInfo\(\)](#). Sie liefert einen Array von dreielementigen String-Arrays zurück, welche die erlaubten Parameter näher beschreiben.

Mittels [getParameter\(String\)](#) können die Werte jedes einzelnen Parameters ausgelesen werden.

```

(1)import java.applet.Applet;
(2)import java.awt.Graphics;
(3)
(4)public class HelloWorldApplet extends Applet {
(5)
(6)    public void paint(Graphics g) {
(7)        g.drawString(    getParameter("displayText"),
(8)                        Integer.parseInt(getParameter("xPos")),
(9)                        Integer.parseInt(getParameter("yPos")) );
(10)    } //paint()
(11)    public String getAppletInfo() {
(12)        return( "written by Mario Jeckle\nversion: 1.1\nCopyright (c) by Mario Jeckle");
(13)    } //getAppletInfo()
(14)    public String[][] getParameterInfo() {
(15)        String appInfo[][] = {
(16)            {"displayText", "anyString", "string to display"},
(17)            {"xPos", "numeric value" , "horizontal position"},
(18)            {"yPos", "numeric value", "vertical position"}
(19)        };
(20)        return appInfo;
(21)    } //getParameterInfo()
(22)} //end class HelloWorldApplet

```

Beispiel 67: Parametrisierung des bekannten HelloWorld-Applets [HelloWorldApplet.java](#)

Applet ausführen

Die Erweiterung des bekannten Beispielapplets zeigt den Einsatz der getAppletInfo-Methode zur Rückgabe Applet-spezifischer Inhalte.

Zusätzlich wird der anzuzeigende Text und seine Position (=Parameter der Methode drawString) über Parameter gesteuert, die durch die Methode getParameterInfo in ihrer Funktion näher beschrieben sind.

Die Parameter werden aus der [\(X\)HTML-Quelle](#) durch die Methode getParameter ausgelesen.

Die Schnittstelle [AppletContext](#) erlaubt die aktive Beeinflussung der Ausführungsumgebung des Applets. Die Methode [getAppletContext](#) liefert die aktuelle Context-Ausprägung zurück.

Jedes AppletContext-Objekt stellt folgende Methoden zur Verfügung:

- [getApplet\(String\)](#)
liefert ein Applet-Objekt zurück, das entsprechend der übergebenen Zeichenkette [benannt](#) ist.
- [getApplets\(\)](#)
Liefert alle Applets im aktuellen Kontext (d.h. dem aktuell angezeigten (X)HTML-Dokument) zurück.
- [getAudioClip\(URL\)](#)
Liefert ein durch URL identifiziertes [AudioClip](#)-Objekt zurück.
- [getImage\(URL\)](#)
Liefert eine über URL identifizierte Graphik als [Image](#)-Objekt zurück.
- [showDocument\(URL\)](#) und [showDocument\(URL, String\)](#)
Laden das durch die URL bezeichnete Dokument, bzw. den bezeichneten Frame, in die aktuelle Ausführungsumgebung, sofern es sich dabei um einen Browser handelt. Appletviewer ignorieren diesen Methodenaufruf.
- [showStatus\(String\)](#)
Zeigt die übergebene Zeichenkette im Statusfenster des Browsers oder Appletviewers an.

```

(1)import java.applet.Applet;
(2)import java.awt.Graphics;
(3)import java.net.URL;
(4)import java.net.MalformedURLException;
(5)
(6)public class HWAURLForward extends Applet {
(7)    public void paint(Graphics g) {
(8)
(9)        g.drawString(    getParameter("displayText"),
(10)                        Integer.parseInt(getParameter("xPos")),
(11)                        Integer.parseInt(getParameter("yPos")) );
(12)        try {
(13)            Thread.sleep(5000); //five seconds delay
(14)        } catch (InterruptedException iee) {
(15)            System.out.println("an InterruptedException occurred");
(16)        } //catch
(17)        try {
(18)            (this.getAppletContext()).showDocument( new URL(getParameter("URL")) );
(19)        } catch (MalformedURLException murle) {
(20)            System.out.println("illegal URL");
(21)        } //catch
(22)    } //paint()
(23)    public String getAppletInfo() {
(24)        return( "written by Mario Jeckle\nversion: 1.1\nCopyright (c) by Mario Jeckle");
(25)    } //getAppletInfo()
(26)    public String[][] getParameterInfo() {
(27)        String appInfo[][] = {
(28)

```

```

(29)         {"displayText", "anyString", "string to display"},
(30)         {"xPos", "numeric value", "horizontal position"},
(31)         {"yPos", "numeric value", "vertical position"},
(32)         {"URL", "any valid URL according to IETF's RFC 1738", "URL to jump to"}
(33)     };
(34)
(35)         return appInfo;
(36)     } //getParameterInfo()
(37)} //class HWAURLForward

```

Beispiel 68: Hello World Applet mit URL-Forward nach fünf Sekunden [HWAURLForward.java](#)

Das Beispiel erweitert das parametrisierte HelloWorld-Applet um eine URL-Weiterleitung nach fünf Sekunden. Die notwendige Zieladresse wird aus dem Parameter URL geladen.

[zugehörige HTML-Seite](#)

Sicherheitskonzept von Applets:

Nachfolgende Tabelle faßt die Rechte und Beschränkungen von Applikationen und Applets, sowohl für die Ausführungsumgebung Web-Browser als auch im Appletviewer, zusammen.

Aktion	Browser	Applet Viewer	Java Applikation, ohne weitere Einschränkungen
Lesen lokaler Dateien		✓	✓
Schreiben lokaler Dateien		✓	✓
Zugriff auf Dateinformation		✓	✓
Löschen von Dateien			✓
Ausführen anderer Programme		✓	✓
Auslesen der Property user.name		✓	✓
Zugriff auf Netzwerkport des Herkunfts-Servers des Applets	✓	✓	✓
Zugriff auf Netzwerkport eines beliebigen Servers		✓	✓
Dynamisches Laden von Java-Bibliotheken		✓	✓
Beenden der virtuellen Maschine mit exit		✓	✓
Erzeugen von Fenstern	✓ mit Warnhinweis	✓	✓
Ausführen nativen-Codes			✓

Das Applet [openProperties](#) zeigt die Inhalte der abfragbaren Systemeigenschaften an.

CannotDisplayApplet

Beim Versuch innerhalb eines Applets eine aus Sicherheitsgründen nicht zugleassene Funktionalität auszuführen wird ein Ausnahmeereignis-Objekt der Klasse [AccessControlException](#) erzeugt.

Hinweis: Die dargestellten Einschränkungen stellen für manche Anwendungsfälle zu deutliche Restriktionen dar. Daher existiert mit den sog. *signed Applets* eine Möglichkeit einzelne Einschränkungen für bestimmte Applets gezielt zu öffnen.

Drei Komponenten wirken bei der Realisierung des Java-Sicherheitsmodells zusammen:

1. Der binäre Bytecode wird nicht direkt durch die physische Hardware ausgeführt, sondern durch die virtuelle Maschine interpretiert.
2. Der Security Manager überwacht alle sicherheitsrelevanten Operationen.
3. Applets mit weitergehenden Berechtigungen müssen sich explizit identifizieren.

Abspielen von Audiodateien:

War diese Funktionalität bis zum Erscheinen der Java 2 Plattform ausschließlich Applets vorbehalten steht sie nun

auch in Java-Applikationen zur Verfügung.

Generelles Vorgehen:

Laden einer Audiodatei mittels der Methodenfamilie [getAudioClip](#) über eine gültige URL.

Im Anschluß kann das rückgegebene Objekt, welches die [AudioClip](#)-Schnittstelle implementiert abgespielt werden.

Java ab Version 2 unterstützt MIDI-Dateien sowie Audiodateien der Formate RMF, WAVE, AIFF und AU.

Zur Soundausgabe in Applikationen wurde die Klassenmethode [newAudioClip\(URL\)](#) eingeführt, die eine Audiowiedergabe auch ohne Existenz eines Applet-Objekts ermöglicht.

Üblicherweise werden im Zusammenhang mit der Audiowiedergabe [Threads](#) verwendet um die Sounds mit mit anderen Ereignissen zu synchronisieren.

Anzeigen von Graphiken:

Bilder in den Formaten GIF, PNG oder JPEG können durch die Methoden [getImage](#) geladen werden.

Die Anzeige erfolgt durch [drawImage\(...\)](#).

Das GIF98a-Animationsformat wird ebenfalls unterstützt.

Anmerkung: Alternativ kann auch signaturgleiche Methode [getImage](#) der Klasse [java.awt.Toolkit](#) benutzt werden.

```
(1)import java.applet.Applet;
(2)import java.awt.Graphics;
(3)import java.awt.GridBagLayout;
(4)import java.awt.Button;
(5)import java.awt.TextField;
(6)import java.awt.Image;
(7)import java.awt.image.ImageObserver;
(8)import java.awt.event.ActionEvent;
(9)import java.awt.event.ActionListener;
(10)
(11)public class GfxViewer extends Applet implements ActionListener {
(12)    protected TextField graphicsName;
(13)    Image imgToDisplay = null;
(14)
(15)    public void init() {
(16)        graphicsName = new TextField ("enter URL of image to view", 50);
(17)        Button loadBtn = new Button ("view");
(18)        GridBagLayout gbl = new GridBagLayout();
(19)        setLayout(gbl);
(20)
(21)        loadBtn.addActionListener(this);
(22)        add(loadBtn);
(23)        add(graphicsName);
(24)    } //init
(25)
(26)    public void actionPerformed(ActionEvent event) {
(27)        imgToDisplay = getImage( getDocumentBase(), graphicsName.getText() );
(28)        repaint();
(29)    } //actionPerformed()
(30)
(31)    public boolean imageUpdate(Image img, int infoflags, int x, int y, int width, int height) {
(32)        repaint();
(33)        if ((infoflags & ImageObserver.SOMEBITS) != 0) {
(34)            System.out.println("part just loaded:"+x+" "+y+" "+ width+" "+height);
(35)            return true;
(36)        } //if
(37)        if ((infoflags & ImageObserver.ALLBITS) != 0) {
(38)            System.out.println("image loaded completely");
(39)            return false;
(40)        } //if
(41)        //gets here when height and width of image is available
(42)        return true;
(43)    } //imageUpdate()
(44)
(45)    public void paint(Graphics g) {
(46)        if (imgToDisplay != null) {
(47)            g.drawImage(imgToDisplay, 1, 1, this);
(48)        } //if
(49)    } //paint
(50)} //class GfxViewer
```

Beispiel 69: Ein einfacher Graphik-Viewer [GfxViewer.java](#)

[Applet ausführen](#)

Das Beispiel implementiert einen einfachen Graphik-Viewer für die beiden standardmäßig unterstützten Formate.

Es verwendet bereits interaktive Komponenten aus dem [Abstract Windows Toolkit](#) (AWT) (siehe [init-](#) und [actionPerformed](#)-Methode). Die rudimentäre Funktionalität erschöpft sich darin, eine Graphikdatei aus dem aktuellen Verzeichnis (ermittelt per [getDocumentBase\(\)](#) zu laden, und im aktuellen Fenster anzuzeigen.

Zunächst wird das Bild per [getImage](#)-Methode referenziert ([graphicsName.getText\(\)](#) gibt den im Textfeld

einggegebenen Namen als String zurück).

Der Aufruf von `repaint()` wirkt auf die gesamte Komponente. Erfolgt das Neuzeichnen, so wird die `paint`-Methode des Applets aufgerufen. Der dort platzierte Methodenaufruf `drawImage` auf dem `Graphics`-Objekt zeichnet das Bild.

Durch Überschreiben der Methode `imageUpdate` (erbt von `Component`, welches die Schnittstelle `ImageObserver` implementiert) kann der Ladevorgang des Bildes verfolgt werden. Diese Methode wird während des Ladevorganges immer wieder automatisch aufgerufen.

Anmerkungen:

- Die `getImage`-Methoden sind asynchron ausgelegt, und kehren daher sofort nach ihrem Aufruf zurück, unabhängig davon ob das referenzierte Bild existiert.
Die erforderlichen Graphikdaten werden erst beim Zeichenversuch in den Speicher geladen.
- Die `drawImage`-Methoden sind (ebenfalls) asynchron ausgelegt, und kehren sofort nach ihrem Aufruf zurück, das Laden des Bildes erfolgt ggf. in einem eigenen Thread.
Hinweis: Die Klasse [MediaTracker](#) stellt eine Implementierung zur Verfügung, welche es ermöglicht das Laden eines Bildes oder einer Audiodatei zu verfolgen.

Animationen:

...sind prinzipiell nichts anderes als die schnelle Wiedergabe verschiedener Einzelbilder. Jedoch gilt es auf einige Randbedingungen Rücksicht zu nehmen:

- Die Anzeigefrequenz der Einzelbilder sollte nicht zu niedrig gewählt sein, sonst erscheint die Animation ruckelig, allerdings auch nicht zu hoch, sonst gehen dem Auge des Betrachters Einzelbilder verloren.
- Beim Neuzeichnen der visuellen Komponente durch die `repaint()`-Methode kann es zu Flimmer- oder Flackereffekten kommen.
- Durch den Darstellungsaufwand sollte der übrige interaktive Programmablauf nicht gestört werden.

Das Beispiel zeigt die Realisierung einer simplen Uhr (Java-[Date](#)-Objekt) als animiertes Applet.

```
(1)import java.applet.Applet;
(2)import java.awt.Graphics;
(3)import java.util.Date;
(4)
(5)public class Clock extends Applet implements Runnable {
(6)    protected int counter;
(7)
(8)    public void start() {
(9)        Thread myClock = new Thread(this);
(10)        myClock.setPriority(Thread.MIN_PRIORITY);
(11)
(12)        myClock.start();
(13)    } //start()
(14)
(15)    public void run() {
(16)        while (true) {
(17)            repaint();
(18)            try {
(19)                Thread.sleep(1000);
(20)            } catch (InterruptedException ie) {
(21)                //ignore it
(22)            } //catch
(23)        } //while
(24)    } //run()
(25)
(26)    public void paint(Graphics g) {
(27)        g.drawString( new Date().toString(),50,50);
(28)    } //paint()
(29)} //class Clock
```

Beispiel 70: Eine einfache Uhr [Clock.java](#)

Applet ausführen

Der notwendige Bildschirmneuaufbau wird in einen separaten niedrig priorisierten Thread verlagert.

Hinweis: Der neue Thread wird nicht, wie bisher per Ableitung von `Thread` definiert und als spezialisiertes Objekt instanziiert. Das Applet implementiert die `Runnable`-Schnittstelle; daher kann ein Objekt vom Typ des Applets einer Variable des Typs `Thread` zugewiesen werden. Erst hierdurch werden die Thread-spezifischen Methoden, wie Prioritätssänderungen, verfügbar.

Die häufigst anzutreffende Realisierung graphischer Animationen ist die Einblendung sich leicht unterscheidender Bilder, wodurch ab einer gewissen Bildfrequenz der Eindruck kontinuierlicher Bewegung entsteht (Trickfilmeffekt). Diese Animationsvariante kann leicht mit den bisher vorgestellten Möglichkeiten realisiert werden. Hierzu werden zunächst die benötigten Bilder vollständig geladen (z.B. [getImage\(String\)](#) oder [getImage\(URL\)](#) aus der [Toolkit](#)-Klasse) und anschließend in schneller Folge am Bildschirm angezeigt (z.B. mit Methoden der [drawImage\(...\)](#)-Familie).

Im Kern ist das Verfahren identisch zur [Anzeige statischer Graphiken](#).


```

(1)import java.awt.Graphics;
(2)import java.awt.Image;
(3)import java.applet.Applet;
(4)
(5)public class AnalogousClock extends Applet implements Runnable {
(6)    static Image images[];
(7)    static int counter=0;
(8)    Thread animator;
(9)
(10)   public void init() {
(11)       images = new Image[12];
(12)       for (int i=0;i<images.length;i++)
(13)           images[i] = getImage( getDocumentBase(), "clock-"+i+".gif");
(14)   } //init()
(15)
(16)   public void start() {
(17)       while ( images[0].getWidth(this) == -1 || images[0].getHeight(this) == -1)
(18)           ;
(19)       this.resize( images[0].getWidth(this) , images[0].getHeight(this) );
(20)       animator = new Thread (this);
(21)       animator.setPriority(Thread.MIN_PRIORITY);
(22)       animator.start ();
(23)   } //start()
(24)
(25)   public void stop() {
(26)       animator = null;
(27)   } //stop()
(28)
(29)   public void run () {
(30)       while (true) {
(31)           try {
(32)               Thread.sleep(200);
(33)           } //try
(34)           catch (InterruptedException ie) {
(35)               //ignore it
(36)           } //catch
(37)
(38)               repaint();
(39)               if (++counter==images.length)
(40)                   counter=0;
(41)           } //while
(42)   } //run()
(43)
(44)   public void paint (Graphics g) {
(45)       System.out.println("drawing image: clock-"+counter+".gif");
(46)       g.drawImage (images[counter], 0, 0, this);
(47)   } //paint()
(48)} //class AnalogousClock

```

Beispiel 71: Analoge Uhr als Bildfolge [AnalogousClock.java](#)

[Applet ausführen](#)

Das Beispielprogramm zeichnet in einem eigenen Thread alle 200 Millisekunden den Bildschirm mit einem neuen Bild neu.

Zunächst werden durch die `init`-Methode die benötigten zwölf Graphiken in einen Image-Array geladen. Zum Ausführungsbeginn (`start`-Methode) wird zunächst die Fenstergröße des Applets (bei Ausführung im Applet-Viewer) auf die Dimensionen der ersten geladenen Graphik -- im Beispiel besitzen alle Graphiken dieselben Ausmaße, daher ist die Auswahl der Referenzgraphik unbedeutend -- gesetzt. Aufgrund der asynchronen Realisierung von `getImage()` muß auf den Abschluß des Ladevorganges gewartet werden; andernfalls liefern die Methoden `getHeight` und `getWidth` mit -1 einen ungültigen Wert zurück.

Der niederprioritäre Thread `animator` sorgt alle 200 Millisekunden für den notwendigen Bildschirmneuaufbau -- durch Aufruf der `repaint`-Methode --, und schaltet die Graphiken fort, wodurch der Animationseffekt entsteht.

Bei dieser Anwendung fällt ein starker **Flimmereffekt** ins Auge. Dieses (unschöne) Verhalten ist oftmals bei graphischen Java-Applets und -Applikationen anzutreffen. Zur Behebung existieren einige Standardlösungsvarianten:

Die einfachste Lösung liegt im Überschreiben der von Component ererbten Methode `update()`.

Im Standardfalle löscht `update()` den gesamten Bildschirmbereich und ruft im Anschluß `paint()` zum vollständigen Neuzeichnen auf.

Im vorliegenden Beispiel ist jedoch das dem Neuzeichnen vorausgehende Löschen nicht notwendig, da die nachfolgende Graphik ohnehin die vorhergehende vollständig überdeckt.

Daher kann die `update`-Methode auf den reinen Aufruf von `paint` beschränkt werden.

Das zusätzliche Codefragment ergibt sich als:

```

public void update(Graphics g) {
    paint(g);
} //update()

```

[ergänzter Quellcode](#)
[ergänzendes Applet ausführen](#)

Diese einfache Maßnahme hilft oftmals bei auftretenden Problemen, jedoch ist sie nur für den eng umrissenen Problemkreis anwendbar, in dem neue graphische Elemente die vorhergehenden vollständig überdecken. Andernfalls kann es zu Darstellungsfehlern kommen.

Daher kommt bei realen Problemstellungen zumeist **double Buffering** zum Einsatz.

Der Mechanismus des double bufferings macht es sich zu Nutze, daß auch im nicht sichtbaren Bereich des Bildschirms graphische Operationen vorgenommen werden können. Die Anwendung verläuft üblicherweise in zwei Schritten: zunächst wird der darzustellende Bildschirminhalt verdeckt vorbereitet, und im zweiten Schritt in einer einzigen -- im Allgemeinen sehr performant ausgeführten -- Kopieroperation in den sichtbaren Bereich übertragen.

Ablaufprinzip:

1. Erzeugung eines `Image`-Objekts beliebiger Größe (gesteuert durch die beiden Übergabeparameter) mittels `createImage(int, int)`.
`createImage(int, int)` ist in der Klasse `Component` definiert, und wird an Applet vererbt.
2. Erzeugung des Zeichenbereichs (*graphics context*) durch die Methode `getGraphics()` auf dem Image-Objekt.
3. Zeichenoperationen auf dem Graphikkontext.
4. Anzeigen des Graphikkontextes innerhalb der `paint`-Methode mit einer Methode aus der `drawImage(...)`-Familie.

Codeschablone:

```
Image im = createImage(100,100);
Graphics graph = im.getGraphics();
//Operationen auf graph

public void paint(Graphics g) {
    g.drawImage(im, ...);
} //paint()
```

In `paint()` werden die vorgezeichneten Image-Objekte ohne weitere Veränderungen angezeigt.

Je nach Komplexität der Anwendung können so viele Image-Objekte wie benötigt parallel gehalten und bei Bedarf angezeigt werden.

▲ 3.2.4 Collection API

Die Collection API liefert als Bestandteil der Java Standard API Hilfsklassen zum Umgang mit generischen Objektsammlungen.

Der Begriff *Collection* bezeichnet Datenstrukturen wie Listen, Stacks, Schlangen, Bäume und Mengen im mathematischen Sinne.

Das **Collection Framework** der Java-API wird aus den vorgegebenen Schnittstellen, den Operationen und ihrer definierten Semantik, den konkreten Implementierungen der wiederverwendbaren Datenstrukturen und der zugrundeliegenden Algorithmen gebildet.

Das wohl bekannteste Beispiel eines solchen Rahmens dürfte die [C++ Standard Template Library](#) (Abk. STL) und die Collection-Klassen der Programmiersprache SmallTalk sein.

Neben dem Collection Framework der Java API existiert mit der [Generic Collection Library for Java](#) (Abk. JGL) auch noch eine frei verfügbare Adaption der STL an Java.

Da Java bis zur Version 1.5 keine parametrische Polymorphie unterstützte sind diese Klassen auch (noch) in ihrer generischen Fassung umgesetzt. Ihr Einsatz unter Nutzung der *Java Generics* wird Gegenstand des [nächsten Kapitels](#) sein.

Um ihre Einsetzbarkeit möglichst universell zu gestalten erlauben alle Collection-Klassen die Verwaltung von Objekten des Typs `Object`, dem gemeinsamen Basistyp aller Java-Objekte.

Durch diese Mimik geht die **Typsicherheit verloren!**

Das bedeutet, daß in jeder Collection ausschließlich Objekte des Typs `Object` verwaltet werden können, die vor dem Einstellen in die Sammlung (implizit und automatisch) in diesen Typ konvertiert werden. Diese Typumwandlung ist als *up-cast* typsicher möglich, da alle Java-Klassen immer direkte oder transitive Subklasse von `Object` sind. Bei der Entnahme von Collection-Elementen ist jedoch die Rekonstruktion des Ursprungstypen notwendig. Diese Typumwandlung ist als *down-cast* explizit anzugeben. Beim Konvertierungsversuch in den falschen Typ kann es hier zu einer `ClassCastException` kommen.

Unter praktischen Gesichtspunkten bietet sich die Vorschaltung einer Typprüfung durch `instanceof` an.

Andererseits schafft die Abkunft aller Java-Objekte von `Object` auch erst die Voraussetzungen für ein generisches Collection Framework zur Verwaltung beliebiger Java-Objekte.

Hiermit sind im Speziellen drei Methoden der Klasse `Object` gemeint:

- [boolean equals\(Object\)](#)
 Erlaubt den Vergleich beliebiger Java-Objekte auf Identität. Daher liefert sie auch bei der Gleichheit aller Attributwerte `false` zurück, da sich die Objekte in ihrer (für den Benutzer nicht sicht- und zugreifbare) Objektidentität unterscheiden.

`equals()` bildet eine Äquivalenzrelation auf `Object` und ist daher reflexiv, symmetrisch und transitiv. Insondere duplikatfreie Sammlungen greifen auf diese Methode zurück, um zu ermitteln, ob sich ein aufzunehmendes Objekt bereits in der Sammlung befindet.

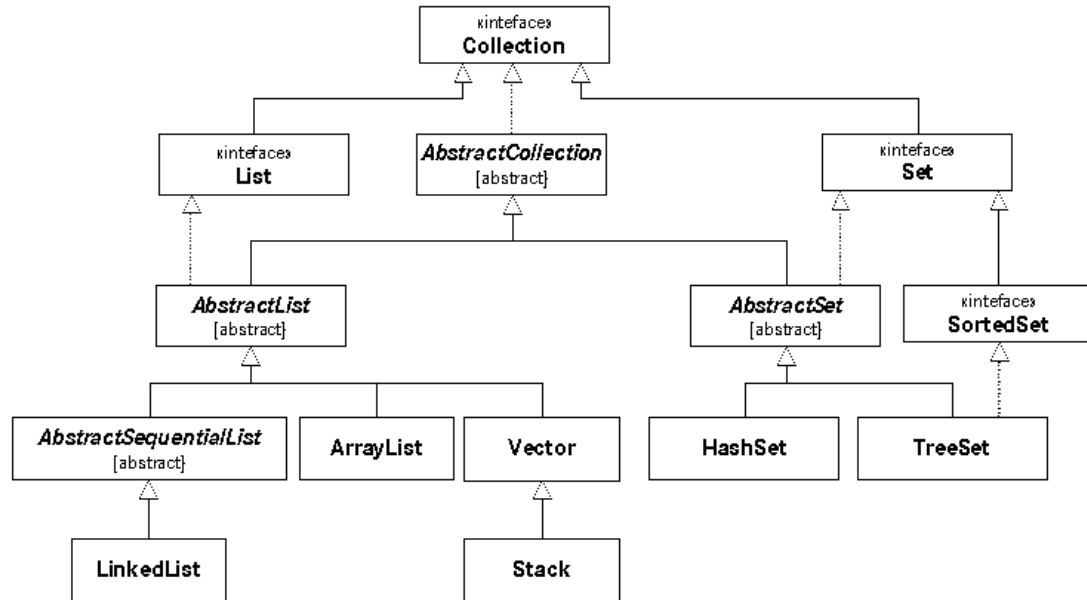
- [`hashCode\(\)`](#)

Die native Methode liefert für einen über die gesamte Laufzeit des Java-Programms unveränderten `int`-Wert als eindeutige Kennung jedes Objekts. Identische Objekte (d.h. Objekte für die `equals` den Wert `true` liefert) haben identische Kennungen.

Die über Hashfunktion jedem Objekt eindeutig zuordenbare `int`-Repräsentation kann als Hash-Code verwendet werden.

Bereits seit Version 1.0 des JDKs stehen die Klassen [Bitset](#), [Dictionary](#), [Hashtable](#), [Stack](#) und [Vector](#) zur Verfügung.

Diese wurde mit dem JDK v1.2 um einige weitere Klassen und Schnittstellen ergänzt.



Mit Ausnahme von `Vector` und `Stack` wurden alle dargestellten Klassen und Schnittstellen mit der Java-2-Plattform eingeführt.

Grundlegende Konzepte:

Gemeinsam sind allen `Collection`-Varianten die basalen Operationen

- Erzeugen der Sammlung
- Einstellen von Elementen
- Entnehmen von Elementen
- Zugriff auf einzelne Elemente
- Löschen der Sammlung

Darüberhinaus bieten die verschiedenen `Collection`-Klassen auch Möglichkeiten zur Suche, Sortierung usw. an.

Interessante (Realisierungs-)Details einer jeden `Collection`-Klasse sind weiter:

- interne Umsetzung
Die tatsächliche technische Realisierung einer wie auch immer präsentierten abstrakten Menge spielt eine entscheidende Rolle hinsichtlich Skalierbarkeit und damit verbundenem Zugriffsverhalten.
- unterstützte Zugriffsmodi
Sie bestimmen wesentlich die praktische Verwendbarkeit einer Sammlung in existierenden Algorithmen. Neben den gebotenen Zugriffsmodi spielt auch ihre Realisierung -- konkret die Anzahl der notwendigen Zugriffe bis das gesuchte Datum gefunden ist -- eine entscheidende Rolle.
- Duplikatfreiheit und Anordnungseigenschaft
Der Sammlungsgedanke an sich unterstellt zunächst weder eine (ungeordnete duplikatfreie) Menge im mathematischen Sinne, noch eine (nach welchen Kriterien auch immer) angeordnete Aufzählung der Elemente.

Das Standard `Collection` Framework umfaßt folgende ausprogrammierte Klassen:

Klasse	Charakteristika / Einsatzgebiete
ArrayList	Dynamischer Array, der intern durch statischen (Java nativen) Array realisiert ist. Lesender und schreibender indizierter Zugriff auf beliebige Elementposition. Ähnelt <code>Vector</code> , ist jedoch nicht threadsicher.
Vector	Bereits im JDK v1.0 enthalten, mittlerweile um Schnittstelle <code>List</code> angereichert. Threadsichere Variante der <code>ArrayList</code> .
LinkedList	Implementierung einer doppelt verketteten Liste. Dadurch konstanter Aufwand beim Einfügen und Löschen an jeder beliebigen Listenposition, jedoch höherer Aufwand beim Anfügen und Entnehmen von letzter Listenposition als bei <code>ArrayList</code> .
Stack	Stack nach dem <i>last-in-first-out</i> -Prinzip. Wie <code>Vector</code> bereits seit dem JDK v1.0 angeboten, und daher nicht originärer Bestandteil des <code>Collection</code> Frameworks. Ebenfalls wie <code>Vector</code> threadsicher realisiert.

Hashtable	Hash-Tabelle die Einstellung von Werten über einen eindeutigen Schlüssel. Bereits mit JDK v1.0 eingeführt; nicht Bestandteil des Collection Frameworks.
HashSet	Realisierung einer Menge im mathematischen Sinne, d.h. anordnungslose (assoziative) Speicherung paarweise verschiedener (Duplikatfreiheit!) Elemente. Gleiches Prinzip wie HashSet, jedoch interne Speicherorganisation als Baum.
TreeSet	Die Elemente werden, sofern nicht anders angegeben, in ihrer <i>natürlichen</i> Ordnung abgelegt. Hierzu muß durch die Klasse jedes abzulegenden Objektes die Comparable -Schnittstelle umgesetzt sein, d.h. die compareTo-Methode implementiert werden.
HashMap	Erweiterung des HashSets um Schlüssel. Ablage von beliebigen Werten (auch Duplikaten) gemeinsam mit einem eindeutigen Schlüssel. (Dieser Strukturtyp wird auch als <i>Bag</i> bezeichnet.) Die Einträge werden intern durch eine Hash-Tabelle verwaltet. Äquivalent zur Hashtable, abgesehen von der mangelnden Threadsicherheit und den zugelassenen Null-Elementen.
TreeMap	Erweiterung der HashMap auf eine Baum-basierte (rot-schwarz-Baum) Ablage eines Wertes (Object) mit einem eindeutigen Schlüssel.
WeakHashMap	Prinzipiell identisch zur HashMap, jedoch werden Schlüssel-Wert-Paare entfernt, sobald die letzte externe Referenz auf den Schlüssel ungültig wird.

Ein einführendes Beispiel:

Die Klasse [Vector](#) ist dem „klassischen“, d.h. bereits mit JDK v1.0 eingeführten, Anteil der Collection-API zuzurechnen. Sie stellt im Grunde eine dynamische Arrayimplementierung zur Verfügung, die intern als statischer Array realisiert ist. Wie bei Arrays üblich erfolgt der Element-Zugriff durch einen ganzzahligen int-Index. Durch den Ausbau der Collection-API in der Java-2-Plattform wurde die Vector-Implementierung nochmals überarbeitet und an die heute Struktur (namentlich die neu eingeführte Schnittstelle [List](#)) adaptiert.

```

(1)import java.io.FileDescriptor;
(2)import java.io.FileNotFoundException;
(3)import java.io.FileReader;
(4)import java.io.IOException;
(5)import java.io.LineNumberReader;
(6)import java.util.Vector;
(7)
(8)public class VectorEx1 {
(9)    public static void main(String[] args) {
(10)        LineNumberReader lnr;
(11)        String line;
(12)        Vector vec = new Vector();
(13)
(14)        try {
(15)            lnr = new LineNumberReader(new FileReader( FileDescriptor.in ) );
(16)
(17)            while ( (line = lnr.readLine()).compareTo("exit")!=0 ) {
(18)                vec.add(line);
(19)            } //while
(20)            lnr.close();
(21)
(22)            for (int i=0; i<vec.size(); i++)
(23)                System.out.println("element no. "+i+": "+vec.get(i) );
(24)        } catch (FileNotFoundException fnfe) {
(25)            System.out.println("cannot find stdin!");
(26)        } catch (IOException ioe) {
(27)            System.out.println("an IOException occurred\n"+ioe.toString()+"\n"+ioe.
getMessage() );
(28)        } //catch
(29)    } //main()
(30)} //class VectorEx1

```

Beispiel 72: Vektor-Operationen [VectorEx1.java](#)

Das Programm liest solange von der Standardeingabe zeilenweise Zeichenketten ein, bis der String *exit* eingegeben wird. Jede eingelesene Zeichenkette wird -- mit der Methode add() -- als Vector-Element angelegt. Nach Beendigung des Einlesevorganges werden die gespeicherten Vector-Elemente in der Reihenfolge der Anlage ausgegeben. Dies geschieht per indiziertem Zugriff auf jedes Element mit der Methode get(int). Der Zugriff erfolgt nicht mehr, wie bei Array üblich per direkter Indizierung in eckigen Klammern, sondern jetzt über eine eigene Methode, die jedoch den gewohnten Sicherheitsmechanismus der ArrayOutOfBoundsException beim Zugriffsversuch außerhalb der Vector-Grenzen beibehält.

Beispielablauf:

```

$java vectorEx1
this
is
a
simple
test
containing an

```

```

empty
line
exit
element no. 0: this
element no. 1: is
element no. 2: a
element no. 3: simple
element no. 4: test
element no. 5: containing an
element no. 6:
element no. 7: empty
element no. 8: line

```

Die **Skalierbarkeit eines Vektor-Objekts** kann durch Konstruktor beeinflusst werden. Der erlaubt die Angabe zweier Freiheitsgrade, der initialen Kapazität und eines Wachstumsfaktors. Werden, wie im Beispiel, keine Parameter übergeben, so wird gegenwärtig (d.h. im JDK v1.3) ein Vector der Größe zehn mit einem Wachstumsfaktor von Null erzeugt.

Wird durch die Einfügemethode `add` die Aufnahmekapazität des Vectors überschritten, so wächst er um den als *capacityIncrement* angegebenen Faktor, oder verdoppelt seine gegenwärtige Größe falls kein Wachstumsfaktor angegeben wurde. In beiden Fällen sind jedoch alle Vector-Elemente in einen neu zu erzeugenden Array zu transferieren, wodurch es bei umfangreichen Vektoren zu Zeitverlusten kommen kann. Die in der gezeigten Implementierung benutzte Umsetzung durch `System.arraycopy` läuft als native Methode jedoch sehr performant ab. Vor grösseren Einfügevorgängen bekannten Umfanges bietet sich jedoch die explizite Allokation des notwendigen Speicherplatzes durch Aufruf der Methode `ensureCapacity(int)` an, die sicherstellt, daß mindestens die im Parameter übergebene Anzahl von Objekten ohne zusätzliche Allokations- und Kopiervorgänge in den Vector eingestellt werden kann.

Der nachfolgende Codeausschnitt aus den Java-Quellen läßt das interne Verhalten von Vector anschaulich werden.

```

package java.util;

public class Vector extends AbstractList implements List, Cloneable, java.io.Serializable {
    protected Object elementData[];
    protected int capacityIncrement;
    protected int elementCount;

    public Vector() {
        this(10);
    } //constructor

    public Vector(int initialCapacity) {
        this(initialCapacity, 0);
    } //constructor

    public Vector(int initialCapacity, int capacityIncrement) {
        ...
        this.elementData = new Object[initialCapacity];
        ...
    } //constructor

    public synchronized Object get(int index) {
        if (index >= elementCount)
            throw new ArrayIndexOutOfBoundsException(index);

        return elementData[index];
    } //get()

    public synchronized boolean add(Object o) {
        ...
        ensureCapacityHelper(elementCount + 1);
        elementData[elementCount++] = o;
        return true;
    } //add()

    private void ensureCapacityHelper(int minCapacity) {
        int oldCapacity = elementData.length;
        if (minCapacity > oldCapacity) {
            Object oldData[] = elementData;
            int newCapacity = (capacityIncrement > 0) ? (oldCapacity + capacityIncrement) :
(oldCapacity * 2);
            if (newCapacity < minCapacity) {
                newCapacity = minCapacity;
            } //if
            elementData = new Object[newCapacity];
            System.arraycopy(oldData, 0, elementData, 0, elementCount);
        } //if
    } //ensureCapacityHelper()

    public Enumeration elements() {
        return new Enumeration() {
            int count = 0;

```

```

    public boolean hasMoreElements() {
        return count < elementCount;
    } //hasMoreElements()

    public Object nextElement() {
        synchronized (Vector.this) {
            if (count < elementCount) {
                return elementData[count++];
            } //if
        } //synchronized
        throw new NoSuchElementException("Vector Enumeration");
    } //nextElement()
}; //elements()
} //class Vector

```

Einen einfacheren Weg zur **Traversierung eines Vectors** bietet die **Schnittstelle Enumeration** an.

Ein solches Objekt wird durch die Methode `elements()` zurückgegeben. Sie abstrahiert nochmals vom indizierten Zugriff, und stellt mit den Methoden `hasMoreElements()` und `nextElement()` generische Zugriffsprimitiven bereit, die so auch in anderen Collection-Klassen Anwendung finden.

Nebenbemerkung: Die Implementierung von `elements()` stellt ein (schönes) Beispiel für die Verwendung anonymer innerer Klassen dar.

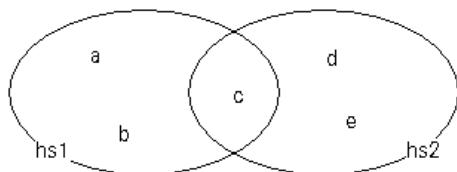
Bei Modifikationen durch nebenläufig ausgeführte Threads am zugrundeliegenden Vector nachdem ein Enumeration-Objekt erzeugt wurde, kann es zu Konsistenzproblemen kommen. Daher wird seit dem JDK v1.2 die Verwendung der Schnittstelle `ListIterator` gegenüber der weiter angebotenen Enumeration empfohlen. Ihre Implementierung ist *fail-fast*, d.h. sie erzeugen ein Ausnahmeereignis-Objekt der Klasse `ConcurrentModificationException` im Falle nebenläufiger Modifikation. Zusätzlich erlaubt der Iterator selbst die Entnahme von Elementen aus der zugrundeliegenden Sammlung.

Der zu einer Collection gehörige Iterator wird über die (von List geerbte) Methode `listIterator()` geliefert.

Vorwärtsreferenz: [Beispiel zur Nutzung eines Iterators](#)

Mengen im mathematischen Sinne (d.h. duplikatfrei und ungeordnet) lassen sich durch die Klasse `HashSet` realisieren.

Mit den (ererbten) Methoden `addAll` (Vereinigung), `retainAll` (Durchschnitt), `removeAll` (Differenz) können die grundlegenden Mengenoperationen realisiert werden.



```

(1)import java.util.HashSet;
(2)import java.util.Iterator;
(3)
(4)public class HashSetEx1 {
(5)    public static void main(String[] args) {
(6)        HashSet hs1 = new HashSet();
(7)        HashSet hs2 = new HashSet();
(8)        HashSet hs3;
(9)
(10)        hs1.add( new String("a") );
(11)        hs1.add( new String("b") );
(12)        hs1.add( new String("c") );
(13)
(14)        hs2.add( new String("c") );
(15)        hs2.add( new String("d") );
(16)        hs2.add( new String("e") );
(17)
(18)        System.out.println("hs1="+hs1.toString() );
(19)        System.out.println("hs2="+hs2.toString() );
(20)
(21)        //union
(22)        hs3 = new HashSet( hs1 );
(23)        hs3.addAll(hs2);
(24)        System.out.println("UNION(hs1,hs2)="+hs3.toString() );
(25)
(26)        hs3 = null;
(27)        //difference
(28)        hs3 = new HashSet( hs1 );
(29)        hs3.removeAll( hs2 );
(30)        System.out.println("hs1 WITHOUT hs2 =" +hs3.toString() );
(31)
(32)        hs3 = null;
(33)        //intersection
(34)        hs3 = new HashSet( hs1 );
(35)        hs3.retainAll( hs2 );
(36)        System.out.println("INTERSECTION(hs1,hs2)="+hs3.toString() );
(37)

```

```

(38)         hs3 = null;
(39)         //symmetric difference
(40)         hs3 = new HashSet( hs1 );
(41)         hs3.addAll( hs2 );
(42)         HashSet tmp1 = new HashSet( hs1 );
(43)         tmp1.retainAll( hs2 );
(44)         hs3.removeAll( tmp1 );
(45)         System.out.println("Symmtetric Difference(hs1,hs2)="+hs3.toString() );
(46)
(47)         hs3 = null;
(48)         //cartesian product
(49)         hs3 = new HashSet();
(50)         String c1;
(51)
(52)         for( Iterator hs1It = hs1.iterator(); hs1It.hasNext(); ) {
(53)             c1 = hs1It.next().toString();
(54)             for( Iterator hs2It = hs2.iterator(); hs2It.hasNext(); )
(55)             {
(56)                 hs3.add( new String( c1 + (hs2It.next()).toString() ) );
(57)             } //for
(58)         } //while
(59)         System.out.println("Cartesian Product(hs1,hs2)="+hs3.toString() );
(60)     } //main()
(61)} //class HashSetEx1

```

Beispiel 73: Die grundlegenden Mengenoperationen [HashSetEx1.java](#)

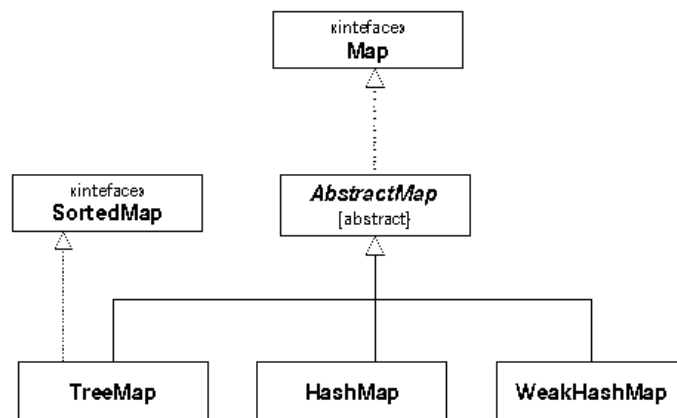
BildschirmAusgabe:

```

hs1=[b, a, c]
hs2=[e, d, c]
UNION(hs1,hs2)=[b, a, e, d, c]
hs1 WITHOUT hs2 =[b, a]
INTERSECTION(hs1,hs2)=[c]
Symmtetric Difference(hs1,hs2)=[b, a, e, d]
Cartesian Product(hs1,hs2)=[ce, cd, cc, be, bd, bc, ae, ad, ac]

```

Das Beispiel zeigt neben Realisierungen der grundlegenden Mengenoperationen auch die Nutzung eines Iterators anstelle der veralteten Enumeration.



Die von der abstrakten Basisklasse [AbstractMap](#) abgeleiteten konkreten Klassen [HashMap](#), [TreeMap](#) und [WeakHashMap](#) stellen eine von der Collection-Hierarchie losgelöste besondere Variante allgemeiner Object-Sammlungen zur Verfügung.

Ihnen allen gemein ist die Speicherung eines eindeutigen Zugriffsschlüssels, zusätzlich zu den Nutzinformationen der verwalteten Objekte.

Die Klasse [HashMap](#) organisiert die abgelegten Objekte intern über eine Hashfunktion, [TreeMap](#) über einen rot-schwarz-Baum.

[WeakHashMap](#) ist prinzipiell identisch zur [HashMap](#), jedoch werden Schlüssel-Wert-Paare entfernt, sobald die letzte externe Referenz auf den Schlüssel ungültig wird.

Das Beispielprogramm ermittelt die Auftrittshäufigkeit von einzelnen Worten oder Zahlen im Eingabestrom. Hierfür wird der bekannte [StreamTokenizer](#) zur entsprechenden Eingabefilterung benutzt.

Grundgedanke des realisierten Algorithmus ist es, das Wort oder die Zahl als Schlüssel in der [HashMap](#) zu nutzen, und die bisher ermittelten Vorkommen als Nutzinformation in einem Integer-Objekt abzulegen.

```

(1)import java.io.StreamTokenizer;
(2)import java.io.FileReader;
(3)import java.io.FileDescriptor;
(4)import java.io.IOException;
(5)import java.util.HashMap;
(6)import java.util.Map;
(7)
(8)public class WordCount {
(9)    private static final Integer ONE = new Integer(1);
(10)
(11)    public static void main(String[] args) {
(12)        Map m = new HashMap();
(13)
(14)        try {
(15)            StreamTokenizer st = new StreamTokenizer( new FileReader( FileDescriptor.
in ) );
(16)            int res;
(17)            Integer freq;
(18)
(19)            while( (res = st.nextToken()) == StreamTokenizer.TT_WORD || res ==
StreamTokenizer.TT_NUMBER) {
(20)                if (res == StreamTokenizer.TT_WORD)
(21)                    freq = (Integer) m.get( st.sval );
(22)            else
(23)                freq = (Integer) m.get( new String(""+st.nval) );
(24)
(25)            if (res == StreamTokenizer.TT_WORD)
(26)                m.put( st.sval, (freq==null ? ONE : new Integer(freq.intValue() + 1)));
(27)            else
(28)                m.put( new String(""+st.nval), (freq==null ? ONE : new
Integer(freq.intValue() +1)));
(29)        } //while
(30)        System.out.println(m.size()+" distinct words detected:");
(31)        System.out.println(m);
(32)    } catch (IOException ioe) {
(33)        System.out.println("an IOException occurred\n"+ioe.getMessage() );
(34)    } //catch
(35)    } //main()
(36)} //class WordCount

```

Beispiel 74: Ermittelt die Auftrittshäufigkeit eines Wortes [WordCount.java](#)

Beispielablauf:

Die Datei [testfile.asc](#):

```

the quick brown fox jumps over the lazy dog
1 2 3 123 1 2 X

```

```

$java wordCount < testfile
13 distinct words detected:
{X=1, lazy=1, fox=1, quick=1, 1.0=2, over=1, the=2, dog=1, 123.0=1, brown=1, 2.0=2, 3.0=1, jumps=1}

```

Die Positionierung eines Elements innerhalb einer der Hash-Sammlungen (HashMap, HashSet und Hashtable) wird während der Ausführung der put-Methode durch eine Hashfunktion, die auf die Methode hashCode der Klasse Object zurückgreift, bestimmt.

Beim Erzeugungsvorgang jedes Hash-...-Objekt kann der Speicherplatzbedarf und das Laufzeitverhalten über zwei Parameter beeinflusst werden. Die *initiale Kapazität* (engl. *initial capacity*) definiert die Anzahl der Einträge in der Hashtabelle, und der *Lastfaktor* (engl. *load factor*) den maximalen Füllungsgrad einer Hashtabelle bevor automatisch eine Kapazitätserweiterung durchgeführt wird.

Werden die beiden Parameter bei der Erzeugung nicht angegeben, so wird vorgabegemäß eine Datenstruktur mit Freiraum für elf Objekte, und einem Lastfaktor von 0.75 erzeugt. Mithin wird die erste Kapazitätserweiterung beim Einfügeversuch des neuen Elements durchgeführt.

Algorithmen auf Objektsammlungen vom Typ Array

... werden durch die Klasse [Arrays](#) in Form statischer Methoden angeboten.

- [asList\(Object\[\]\)](#)
Erzeugt eine Liste fester Größe aus einem Array beliebiger Objekte.
- [binarySearch\(...\)](#)
Stellt für die verschiedenen [Primitivtypen](#)-Arrays und allgemeine Object-Arrays eine Suche nach beliebigen Inhaltswerten zur Verfügung.
- [equals\(...\)](#)
Vergleicht zwei Arrays von identischem Typ auf inhaltliche Gleichheit.
- [fill\(...\)](#)
Ersetzt alle Elemente eines Arrays durch das Übergebene.
- [sort\(...\)](#)
Sortiert übergebenen Array mit einer modifizierten Variante des Quicksort-Algorithmus.

Algorithmen auf Objektsammlungen vom Typ List

... werden durch die Klasse [Collections](#) in Form statischer Methoden angeboten.

- [binarySearch\(...\)](#)
Führt eine binäre Suche nach einem gegebenen Wert in einer Listen-artigen (d.h. Collection-Klasse welche die Schnittstelle List implementiert) durch.
Die gegenwärtige Implementierung nimmt jedoch AbstractSequentialLists und deren Subklassen (in der Standard-API realisiert: LinkedList) explizit von dieser Mimik aus, und realisiert für diese Sammlungstypen eine sequentielle Suche.
- [copy\(List, List\)](#)
Kopiert Listenelemente in eine andere Liste.
- [Enumeration\(Collection\)](#)
Liefert für jeden Collection-Typ eine Enumeration zur Traversierung zurück.
Anmerkung: Leider existiert derzeit keine Äquivalent das die modernere und sicherere Variante Iterator zurückliefern würde.
- [fill\(List, Object\)](#)
Ersetzt alle Elemente einer Liste durch das gegebene Object.
- [min\(...\)](#) und [max\(...\)](#)
Liefen das Minimum, bzw. Maximum, einer Liste, ggf. unter Anwendung einer speziellen Vergleichsoperation. Die Suche wird nach der Methode des sequentiellen Schwellenwertes durchgeführt.
- [nCopies\(Object\)](#)
Liefert eine Liste mit n Kopien eines bestimmten Objekts.
- [reverse\(List\)](#)
Kehrt die Elementordnung für eine Liste um.
- [reverseOrder\(\)](#)
Liefert ein [Comparator](#)-Objekt zurück welches die Anordnung der Liste umkehrt.
- [shuffle\(List\)](#)
Permutiert Objekte der Liste.
- [sort\(...\)](#)
Sortiert Liste nach einem modifizierten Mergesort-Verfahren.
Die Verwendung des Mergesort-Algorithmus ist nicht zwingend, API-Implementierungen können ihn gegen durch Algorithmen ersetzen. Einzige Bedingung hierbei ist die Stabilität. D.h. die Positionserhaltung gleicher Elemente; sie sollen durch den Sortiervorgang nicht verschoben werden.
In der Implementierung wird zunächst aus der Liste in ein Object-Array erzeugt, auf dem dann die sort-Methode der Klasse Arrays aufgerufen wird, die den Mergesort dann für Objects implementiert.
- [synchronizedCollection\(Collection\)](#), [synchronizedList\(List\)](#), [synchronizedMap\(Map\)](#), [synchronizedSet\(Set\)](#), [synchronizedSortedMap\(SortedMap\)](#) und [synchronizedSortedSet\(SortedSet\)](#)
Liefen eine threadsichere Variante der jeweiligen Eingabedatenstruktur zurück.
- [unmodifiableCollection\(Collection\)](#), [unmodifiableList\(List\)](#), [unmodifiableMap\(Map\)](#), [unmodifiableSet\(Set\)](#), [unmodifiableSortedMap\(SortedMap\)](#) und [unmodifiableSortedSet\(SortedSet\)](#)
Liefen eine unveränderbare Sicht auf die jeweilige Eingabedatenstruktur zurück.

Anmerkung zur Verwendung der beiden Sortierv Verfahren *Mergesort* und *Quicksort* in den Klassen Arrays und Collections:

Beim ersten Hinsehen verwundert die Implementierung zweier verschiedener Sortierv Verfahren ein wenig, nicht zuletzt wegen des Wechsels des Sortierv Verfahrens in der Implementierung der sort-Methoden der Klasse Arrays. Wird dort doch für alle Sortieroperationen auf Primitivtypen-Arrays eine modifizierte Quicksort-Variante eingesetzt, jedoch für Object-Arrays zu Mergesort übergegangen.

Dies liegt in der Natur der beiden Sortierv Verfahren begründet. Quicksort ist eines der bekanntesten und vermutlich auch am besten erforschtesten Sortierv Verfahren; mit guten Leistungsdaten im allgemeinen Falle. Jedoch kann der Sortiervorgang im schlechtesten Falle bis zu n^2 Schritte benötigen. Insbesondere für große n ist daher Mergesort -- mit seinen garantierten $n \log(n)$ Schritten effizienter.

Allerdings benötigt Quicksort deutlich weniger Speicherplatz (eigentlich nur den Stack durch die rekursiven Aufrufe), während Mergesort für seinen Ablauf stets den doppelten Speicherbedarf der zu sortierenden Datenstruktur für sich reklamiert.

Aus praktischen Gründen, nicht zuletzt wegen des uniformen Laufzeitverhaltens, wurde dem speicherplatzintensiveren Mergesort der Vorzug gegeben. Da in realen Anwendungen in den seltensten Fällen reine Primitivtypen-Arrays sortiert werden, nutzt der Anwender überwiegend die vorhandenen Mergesort-Implementierungen, die jedoch in (begründeten) Einzelfällen durchaus durch eigene Routinen ersetzt werden können.

 3.2.5 Parametrische Polymorphie / Generics

Die wirkungsmächtigste Neuerung der Java Version 1.5 bildet die Einführung der sog. *Generics*, welche das Feld der Template-basierten Metaprogrammierung und mithin die parametrische Polymorphie eröffnen.

Das sinnvollste Einsatzfeld dieses Mechanismus bildet die Anwendung auf die Vorhandenen Klassen der Collection-API zur Realisierung typsicherer Objektsammlungen.

Die allgemeine Syntax der Generics erfordert die Nachstellung, des durch spitze Winkelklammern eingeschlossenen Typnamen, nach dem Namen der so typisierten Sammlung. So wird eine Liste, die ausschließlich Objekte des Typs String enthält als `List<String>` definiert.

Als Resultat einer so definierten Sammlungsklasse wird bereits zum Übersetzungszeitpunkt die Konformität abgeprüft. Daher wird bei jedem Versuch Daten in eine solch konkretisierte Sammlung einzufügen geprüft, ob diese

unter Beachtung der Typrestriktion kompatibel zum in der Deklaration angegebenen Inhaltstyp sind.

Konventionsgemäß erlaubt Java ausschließlich die Festlegung von Klassen als Inhaltstypen von Objektsammlungen; Primitivtypen sind von der Verwendung ausgeschlossen.

Das nachfolgende Beispiel zeigt die Definition einer Sammlung des Typs `LinkedList`, deren Inhaltselemente auf Ausprägungen des Typs `Integer` beschränkt sind.

```
(1)import java.util.LinkedList;
(2)
(3)public class GenTest1 {
(4)    public static void main(String[] args) {
(5)        LinkedList<Integer> integerList = new LinkedList<Integer>();
(6)
(7)        integerList.add(new Integer(42));
(8)        integerList.add( 84 );
(9)
(10)        System.out.println( integerList );
(11)    } //main()
(12)} //class GenTest1
```

Beispiel 75: Typisierung einer generischen Liste [GenTest1.java](#)

Das Beispiel zeigt die Definition einer typisierten `LinkedList`, für die bereits zum Übersetzungszeitpunkt geprüft wird, daß ausschließlich Ausprägungen der Klasse `Integer` der Liste hinzugefügt werden.

Diese Restriktion verhindert jedoch, vermöge der Nutzung des [dynamischen Boxings](#) nicht, daß Ausprägungen des Primitivtyps `int` direkt übergeben werden, da diese für den Programmierer transparent in Objekte des Type `Integer` umgewandelt werden.

Versuche Objekte oder Werte anderen Typs in die Liste aufzunehmen werden bereits zum Übersetzungszeitpunkt erkannt und als Fehler gemeldet.

Die Einführung typisierter Sammlungen bedingt auch die entsprechende Adaption der Hilfsklassen zur Traversierung. vor diesem Hintergrund zeigt das nachfolgende Beispiel die Typisierung eines Iterators. Als Konsequenz dieser Konkretisierung des Iterators liefert der Aufruf der Methode [next](#) statt der generischen `Object`-Instanz eine Ausprägung des Parametertyps `String`.

```
(1)import java.util.LinkedList;
(2)import java.util.Iterator;
(3)
(4)public class GenTest2 {
(5)    public static void main(String[] args) {
(6)        LinkedList<String> nameList = new LinkedList<String>();
(7)
(8)        nameList.add( "Berta" );
(9)        nameList.add( "Anton" );
(10)       nameList.add( "Cesar" );
(11)
(12)       Iterator<String> listIterator = nameList.iterator();
(13)       String value;
(14)       while(listIterator.hasNext()) {
(15)           value = listIterator.next();
(16)           System.out.println( value );
(17)       } //while
(18)    } //main()
(19)} //class GenTest2
```

Beispiel 76: Typisierung eines Iterators [GenTest2.java](#)

Neben der Anwendung auf bestehende Klassen des Collection-Frameworks können Generics auch für eigenerstellte Klassen eingesetzt werden.

Das Beispiel zeigt die Definition der Klasse `TopTen`, welche eine Liste von Einträgen beliebigen Typs zusammen mit einer Platzierung gestattet. Der Inhaltstyp wird hierbei als Polymorphieparameter zum Konstruktionszeitpunkt der `TopTen`-Objekte festgelegt. Innerhalb der Klasse wird zur Verwaltung der Daten eine `TreeMap` eingesetzt, die entsprechend der Initialisierung des `TopTen`-Objekts erzeugt wird um auch auf dieser Ebene die Typkonformität zu gewährleisten.

```
(1)import java.util.TreeMap;
(2)import java.util.Collection;
(3)import java.util.Iterator;
(4)
(5)public class GenTest3 {
(6)    public static void main(String[] args) {
(7)        TopTen<String> music = new TopTen<String>();
(8)        music.addAtPosition(1, "St. Anger/Metallica");
(9)        music.addAtPosition(3, "20 Jahre - Nena feat. Nena/Nena");
(10)       music.addAtPosition(2, "9/Eros Ramazotti");
(11)
(12)       music.print();
(13)    } //main()
```

```

(14)} //class GenTest3
(15)
(16)class TopTen<Type> {
(17)    private TreeMap<Integer,Type> content;
(18)    public TopTen() {
(19)        content = new TreeMap<Integer,Type>();
(20)    } //constructor
(21)    public boolean addAtPosition(int pos, Type value) {
(22)        if (content.size() == 10) {
(23)            return false;
(24)        } else {
(25)            content.put(new Integer(pos), value);
(26)            return true;
(27)        } //else
(28)    } //addAtPosition()
(29)    public Type getFromPosition(int pos) {
(30)        return content.get(new Integer(pos));
(31)    } //getFromPosition()
(32)    public void print() {
(33)        Iterator<Type> i = content.values().iterator();
(34)        while (i.hasNext()) {
(35)            System.out.println( i.next() );
(36)        } //while()
(37)    } //print()
(38)} //class TopTen

```

Beispiel 77: Anwendung von Generics für eigene Klassen [GenTest3.java](#)

Neben der Angabe von Klassen zur Typisierung einer generischen Aufzählung können, mit derselben Mächtigkeit, auch Aufzählungstypen angegeben werden. Das nachfolgende Beispiel zeigt eine Anwendung:

```

(1)import java.util.LinkedList;
(2)import java.util.HashMap;
(3)
(4)public class EnumTest4 {
(5)public static void main(String argv[]) {
(6)    enum DayOfWeek {monday, tuesday, wednesday, thursday, friday,
(7)        saturday, sunday};
(8)
(9)    LinkedList<DayOfWeek> allDays = new LinkedList<DayOfWeek>();
(10)    allDays.add(DayOfWeek.monday);
(11)    allDays.add(DayOfWeek.tuesday);
(12)    allDays.add(DayOfWeek.wednesday);
(13)    allDays.add(DayOfWeek.thursday);
(14)    allDays.add(DayOfWeek.friday);
(15)    allDays.add(DayOfWeek.saturday);
(16)    allDays.add(DayOfWeek.sunday);
(17)
(18)    HashMap<DayOfWeek,Double> sales = new HashMap<DayOfWeek,Double>();
(19)
(20)    sales.put(DayOfWeek.monday, 3000.0 );
(21)    sales.put(DayOfWeek.tuesday, 5000d );
(22)    sales.put(DayOfWeek.thursday, 7500D );
(23)    sales.put(DayOfWeek.wednesday, (double) 8000 );
(24)    sales.put(DayOfWeek.friday, 55000.0 );
(25)    sales.put(DayOfWeek.saturday, new Double("10000"));
(26)    sales.put(DayOfWeek.sunday, (new Double("0")).doubleValue() );
(27)
(28)    for (DayOfWeek w : allDays)
(29)        System.out.println(w+" "+sales.get(w));
(30)    } //main()
(31)} //class EnumTest4

```

Beispiel 78: Aufzählungstypen als Polymorphieparameter [EnumTest4.java](#)

Zwar vermeidet der Einsatz der Generics die Anwendung unsicherer expliziter Typkonversionen fast völlig, jedoch ist ihre Notwendigkeit in Einzelfällen nach wie vor gegeben.

Das nachfolgende Beispiel zeigt einen solchen Fall. Es definiert eine als Vector angelegte Sammlung von Objekten des Typs Person. Gemäß der Substitutionsregeln können auch Spezialisierungen der Person, mithin Ausprägungen der davon abgeleiteten Klasse Mitarbeiter, typkonform der Sammlung hinzugefügt werden.

Da die Methode get im allgemeinen Falle Ausprägungen der Klasse Object und unter Einsatz der Generics Ausprägungen des Parametertyps (in diesem Falle: Person) liefert würde die abgespeicherte Mitarbeiter-Instanz lediglich als Ausprägung der Klasse Person aus der Sammlung entnommen werden. Um eine korrekte Weiterverarbeitung (z.B. in Form des Zugriffs auf das Attribut personalnummer) als Mitarbeiter zu gewährleisten ist daher eine explizite Typkonversion notwendig.

```

(1)import java.util.Vector;
(2)
(3)public class GenTest4 {
(4)    public static void main(String[] args) {
(5)        Vector<Person> persL = new Vector<Person>();
(6)
(7)        Person p = new Person();
(8)        p.name = "Max Mustermann";
(9)        persL.add(p);
(10)
(11)        Mitarbeiter m = new Mitarbeiter();
(12)        m.name = "John Doe";
(13)        m.personalnummer = "1234";
(14)        persL.add(m);
(15)
(16)        Person p2 = persL.get(0);
(17)        System.out.println( p2.name );
(18)
(19)        Mitarbeiter m2 = (Mitarbeiter) persL.get(1);
(20)        System.out.println( m2.name );
(21)        System.out.println( m2.personalnummer );
(22)    } //main()
(23)} //class GenTest4
(24)
(25)class Person {
(26)    public String name;
(27)} //class Person
(28)
(29)class Mitarbeiter extends Person {
(30)    public String personalnummer;
(31)} //class Mitarbeiter

```

Beispiel 79: Explizite Typwandlung trotz Generics [GenTest4.java](#)

▲ 3.2.6 Reflection API

Mit der JDK-Version 1.1 wurde die Möglichkeit geschaffen zur Laufzeit Informationen über die verwendeten Informationsstrukturen zu gewinnen. Genaugenommen handelt es sich hier um Information über Information, die üblicherweise als *Metainformation* bezeichnet wird.

Bereits seit der Version 1.0 existierte mit der Klasse [Class](#) ein Mechanismus, der Möglichkeit zur Ermittlung des Typs im Stile der aus C++ bekannten *Runtime Type Identification* bietet.

Der **Begriff Laufzeittyp** meint den konkreten Typen eines Attributes oder einer Variable (im Allgemeinen: einer referenzierten typisierten Speicherposition). Dieser Typ muß nicht zwingend über die gesamte Ausführungszeit mit dem zum Definitionszeitpunkt festgelegten identisch sein. Er kann sich durch typkonforme Neuzuweisung ändern.

Die Klasse Class

Zur *Motivation*: [Bisher](#) war es uns mit dem instanceof-Operator nur möglich zu entscheiden ob ein gegebenes Objekt Ausprägung einer bestimmten Klasse ist. Hierfür mußte der Namen der Klasse zum Ausführungszeitpunkt des Operators bekannt sein.

Die Methode [getClass\(\)](#) welche, da von Object geerbt, auf allen Java-Objekten zur Verfügung steht behebt dieses Manko.

Sie liefert ein Objekt der Klasse [Class](#) zurück welches -- neben zahlreichen weiteren Informationen -- den Namen der Klasse zur Laufzeit verfügbar werden läßt.

Class ist im Paket java.lang angesiedelt, und steht daher ohne explizite import-Anweisung in allen Java-Programmen zur Verfügung. Zu jeder Klasse im System existiert je ein solches Class-Objekt.

```

(1)public class RefEx1 {
(2)    public static void main(String[] args) {
(3)        Person p1 = new Person();
(4)        Mann m1 = new Mann();
(5)
(6)        System.out.println("runtime class of variable p1="+ p1.getClass().getName() );
(7)        System.out.println("runtime class of variable m1="+ m1.getClass().getName() );
(8)
(9)        p1 = m1; //up-cast is typesafe
(10)       System.out.println("runtime class of variable p1="+ p1.getClass().getName() );
(11)    } //main()
(12)} //class RefEx1
(13)
(14)class Person {

```

```

(15)} //class Person
(16)
(17)class Mann extends Person {
(18)} //class Mann
(19)
(20)class Frau extends Person {
(21)} //class Frau

```

Beispiel 80: Ermittlung der Laufzeitklasse [RefEx1.java](#)

Bildschirmausgabe:

```

$java RefEx1
runtime class of variable p1=Person
runtime class of variable m1=Mann
runtime class of variable p1=Mann

```

Die Anwendung definiert neben der Hauptklasse drei weitere Klassen Person, Mann und Frau. Sie sind in der Weise organisiert, daß Mann und Frau als Subklassen von Person realisiert sind.

Zunächst werden Variablen vom Typ Person und Mann definiert und mit Ausprägungen der Definitionstypen initialisiert. Durch den Methodenaufruf `getName()` auf dem durch `getClass()` zurückgelieferten Class-Objekt kann der Name der Laufzeitklasse im Klartext ermittelt werden. So liefert die Variable `p1` welche auf eine Person-Ausprägung verweist erwartungsgemäß den Namen dieser Klasse zurück; für die Variable `m1` und die Klasse Mann gilt dies analog.

Durch der Zuweisung des Mann-Objekts `m1` an die Variable `p1` (vom Typ Person) verändert sich deren Laufzeittyp zu Mann.

Im vorangegangenen Beispiel wird ein Objekt zur Ermittlung der zugehörigen Laufzeitklasse benutzt. Ist der Klassenname bekannt, so kann durch den Methodenaufruf `forName(String)` das zur Klasse mit dem übergebenen Namen gehörige Class-Objekt ermittelt werden.

```

(1)public class RefEx2 {
(2)    public static void main(String[] args) {
(3)        try {
(4)            Class c1 = Class.forName("Person");
(5)            System.out.println("Name of class:"+c1.getName() );
(6)        } catch (ClassNotFoundException cnfe) {
(7)            System.out.println("cannot find class\n"+cnfe.toString()+"\n"+cnfe.
getMessage() );
(8)        } //catch
(9)    } //main()
(10)} //class RefEx2
(11)
(12)class Person {
(13)} //class Person

```

Beispiel 81: Ermittlung eines Class-Objekts über den Klassennamen [RefEx2.java](#)

Bildschirmausgabe:

```

$java RefEx2
Name of class:Person

```

Die Klasse Person wird geladen, und per `getName()` ihr Name -- erwartungsgemäß Person -- ausgegeben.

Kann die Klasse nicht gefunden werden, so wird ein [ClassNotFoundException](#) Ausnahmeereignis-Objekt erzeugt.

Um unnötigen Tippaufwand zu sparen existiert mit `T.class` eine abkürzende Schreibweise für alle Javatypen `T` aus dem [Typsystem](#).

Beispiel:

```

Class c1 = Person.class;
Class c2 = int.class;
Class c3 = Double[].class;

```

Hinweis: Aus historischen Gründen liefert die Methode `getName()` für alle Array-Typen eine etwas unschöne abkürzende Syntax, die sich am [im Class-File verwendeten Format](#) orientiert:

Zunächst das Präfix [für jede Array-Schachtelung, gefolgt von ...

B für byte

C für char

D für double

F für float

I für int

J für long

L; für eine Klasse oder Schnittstelle

S für short

Z für boolean.

so liefert `(new int[3][4][5][6][7][8][9]).getClass().getName()` als Namen die Zeichenkette `[[[[[[[I` zurück.

Die **Erzeugung neuer Objekte aus einem existierenden Class-Objekt** erfolgt durch die Methode [newInstance\(\)](#). Sie ruft intern den parameterlosen Vorgabekonstruktor der durch das Class-Objekt vertretenen Klasse auf. Die `newInstance`-Methode unterstützt keine Aufrufe parametrisierter Konstruktoren. Für diesen Anwendungsfall sollte auf die Klasse `Constructor` der (modernerer) Reflection API zurückgegriffen werden.

```
(1)public class RefEx3 {
(2)    public static void main(String[] args) {
(3)        Person p1 = new Person(); //normal object creation
(4)        p1.sayHello();
(5)
(6)        try {
(7)            Person p2 = (Person) Class.forName("Person").newInstance();
(8)            p2.sayHello();
(9)        } catch (ClassNotFoundException cnfe) {
(10)            System.out.println("cannot find class\n"+cnfe.toString()+"\n"+cnfe.
getMessage() );
(11)        } catch (InstantiationException ie) {
(12)            System.out.println("an InstantiationException occured\n"+ie.toString()
+"\n"+ie.getMessage() );
(13)        } catch (IllegalAccessException iae) {
(14)            System.out.println("an IllegalAccessException occured\n"+iae.toString()
+"\n"+iae.getMessage() );
(15)        } //catch
(16)    } //main()
(17)} //class RefEx3
(18)
(19)class Person {
(20)    public void sayHello() {
(21)        System.out.println("hello from person!");
(22)    } //sayHello()
(23)} //class Person
```

Beispiel 82: Erzeugung eines neuen Objekts mit der Methode `newInstance` [RefEx.java](#)

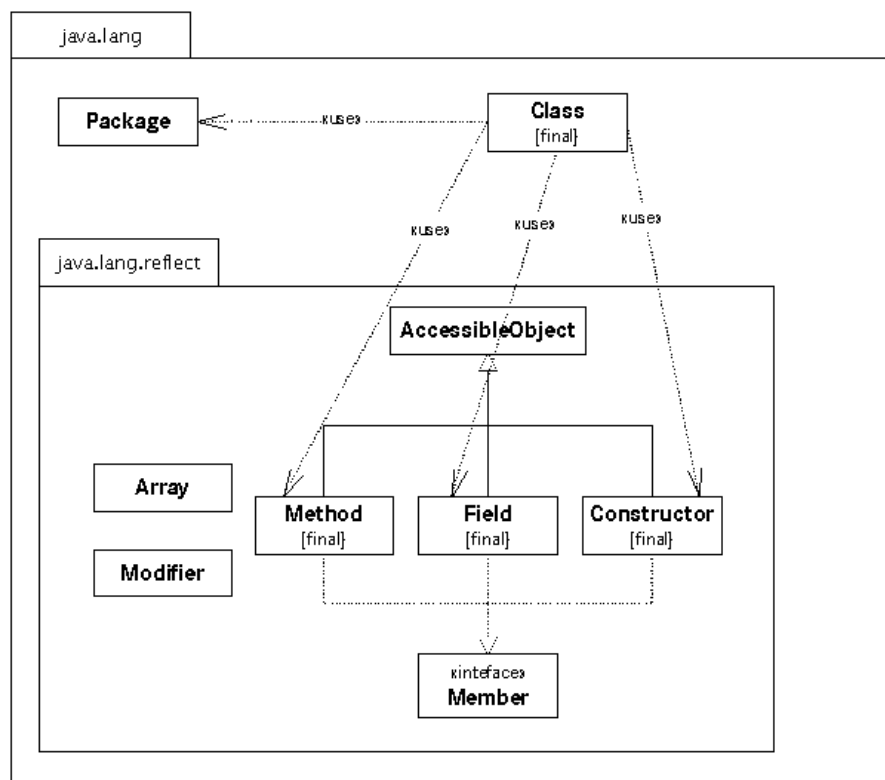
Das Programm gibt zweimal die Zeichenkette *hello from person!* am Bildschirm aus. Die Erzeugung des zweiten Personen-Objektes (p2) erfolgt durch die Kopplung der Methoden `forName` und `newInstance`. Letztere liefert eine Ausprägung der Klasse `Object` zurück, weshalb -- um die Methode `sayHello` der davon abgeleiteten Klasse `Person` aufrufen zu können -- die explizite Typumwandlung erfolgen muß.

Ferner stellt die Klasse `Class` weitere Methoden zur Untersuchung der Charakteristika definierter Klassen bereit:

- [getSuperclass\(\)](#)
Liefert die Superklasse einer gegebenen Klasse. Für die Klasse `java.lang.Object`, Schnittstellen, alle Primitivtypen und `void` wird null zurückgeliefert.
- [getInterfaces\(\)](#)
Liefert die durch eine Klasse direkt implementierten Schnittstellen; ererbte Schnittstellen werden nicht geliefert.
- [isInterface\(\)](#)
`true` falls das Class-Objekt eine Schnittstelle ist, andernfalls `false`.
- [isInstance\(Object\)](#)
Entspricht dem `instanceof`-Operator.
- [isAssignableFrom\(Class\)](#)
Prüft ob die Zuweisung des als Parameter übergebenen Klassentypen an das Objekt auf dem diese Methode aufgerufen wird möglich ist.
Die Methode bietet damit direkten Zugriff auf den JVM-Opcode [checkcast](#); ohne jedoch bei Unmöglichkeit der Zuweisung ein Ausnahmeereignis-Objekt zu generieren.

Das Reflection-Paket

Wie bereits angedeutet wurde die Klasse `Class` mit Einführung der Reflection API in der Java Version 1.1 stark erweitert. Sie schlägt -- über die Rückgabetypen der neu geschaffenen Methoden -- eine Brücke zu den Klassen des Reflection-Paketes [java.lang.reflect](#).



Die Reflection-API ist als Subpaket von `java.lang` organisiert. Ihre Klassen dienen lediglich zur Inspektion existierender Klassen einer Java-Laufzeit-Umgebung. Daher sind die Konstruktoren der von `AccessibleObject` abgeleiteten Klassen, sowie der der Klasse `Array`, als `private` deklariert; Ausprägungen dieser Klassen können ausschließlich durch die virtuelle Maschine erzeugt werden.

Im Folgenden werden zunächst die Methoden zur Ermittlung verschiedenster Informationen am [konkreten Codebeispiel](#) vorgestellt.

Die verschiedenen Methoden zur Einflußnahme auf bereits erzeugte Metainformationsobjekte werden [im Anschluß](#) diskutiert.

Ermittlung Klassen- und Schnittstellen-bezogener Information

Klassen und Schnittstellen werden durch die Reflection-API gleichbehandelt. Die Unterscheidung kann durch Methode `isInterface` auf jedem Class-Objekt getroffen werden.

Ferner stehen zur Verfügung:

- **Modifier** -- `getModifiers`
Der Aufruf liefert die verschiedenen Details als `int`-Wert, der die gesetzten Modifier Bit-codiert enthält. Zur Entschlüsselung der einzelnen Modifier sollten die Methoden der Reflection-Klasse `Modifier` eingesetzt werden, um auch hinsichtlich zukünftiger Java-Spracherweiterungen stabilen Code zu produzieren.
[Aufruf im Beispiel](#)
Die [Umsetzung im Beispiel](#) nutzt die Implementierung der `toString()`-Methode der Klasse `Modifier`, welche die in der [Java Language Specification](#) definierte kanonische Repräsentation zurückliefert.
- **Komponententyp-Ermittlung** -- `getComponentType()`
Ermöglicht die Ermittlung des zugrundeliegenden Typen falls es sich bei der Klasse um einen Array handelt. Bei geschachtelten Arrays ist diese Methode u.U. mehrfach anzuwenden.
[Verwendung im Beispiel](#)
- **Intern deklarierte Klassen und Schnittstellen** -- `getDeclaredClasses()`
Liefert die deklarierten (nicht anonymen) [inneren Klassen](#) und inneren Schnittstellen als Class-Array zurück.
[Verwendung im Beispiel](#)
- **Name** -- `getName()`
Liefert den vollqualifizierten Namen der Klasse oder Schnittstelle.
[Verwendung im Beispiel](#)
- **Paketzugehörigkeit** -- `getPackage`
Liefert das definierende Paket der Klasse als Ausprägung der Klasse `Package`. Bei Klassen im Vorgabepaket, d.h. ohne explizite package-Definition, wird `null` zurückgegeben.
[Verwendung im Beispiel](#)

Ermittlung Attribut-bezogener Informationen

Der Zugriff auf Attribute und Attribut-spezifische Informationen -- in der Java-Terminologie Felder genannt -- (wie Namen, Sichtbarkeitsbereich und Typ) wird durch die Methoden `getField(String)`, für ein namentlich bekanntes Feld, und `getFields()` für *alle* Attribute einer Klasse, realisiert. In beiden Fällen ist wird genau ein, bzw. ein Array von Objekten der Reflection-Klasse `Field` zurückgeliefert.

Beide Methoden berücksichtigen dabei *alle* Attribute/Felder einer Klasse, einschließlich der von der Superklasse und deren Superklassen ererbten. Zur Ermittlung der direkt auf dem betrachteten Class-Objekt definierten Attribute können Methoden `getDeclaredField(String)` bzw. `getDeclaredFields()` eingesetzt werden.

Verwendung im Beispiel

Field-Objekte bieten Zugriff auf die verschiedenen deklarativen Charakteristika eines Attributs:

- Modifier ([siehe Kapitel 2](#)) -- [getModifiers\(\)](#).
Der Aufruf liefert die verschiedenen Details als int-Wert, der die gesetzten Modifier Bit-codiert enthält. Zur Entschlüsselung der einzelnen Modifier sollten die Methoden der Reflection-Klasse [Modifier](#) eingesetzt werden, um auch hinsichtlich zukünftiger Java-Spracherweiterungen stabilen Code zu produzieren.
Die [Verwendung im Beispiel](#) nutzt die Implementierung der [toString\(\)](#)-Methode der Klasse [Modifier](#), welche die in der [Java Language Specification](#) definierte kanonische Repräsentation zurückliefert.
- Typ (einer der [Primitivtypen](#), ein Array, eine beliebige zugreifbare Klasse oder eine Schnittstelle) -- [getType\(\)](#).
Die Methode liefert ein Objekt vom Typ [Class](#) zurück, auch wenn die Primitivtypen nicht als (first class) Objekte realisiert sind.
Zur Ermittlung des tatsächlichen Typs sind weitere Operationen auf dem [Class](#)-Objekt notwendig. [isArray\(\)](#) klärt ob es sich bei dem Attributtypen um einen Array-Typ handelt. Da Java mehrdimensionale Arrays als geschachtelte eindimensionale Pendants realisiert muß in einem solchen Falle per [getComponentType\(\)](#) weiternavigiert werden.
Diese Methode liefert wiederum ein [Class](#)-Objekt zurück.
Der vollqualifizierte (d.h. im Falle der Existenz um den durch das Paket gebildeten Namensraum angereicherte) Name eines Typen wird durch [getName\(\)](#) zurückgeliefert.
Die Methode [printAttribute](#) des Beispiels zeigt die Ermittlung des Typs durch die vorgestellten Methoden im Überblick.
- Name -- [getName\(\)](#).
Liefert den Attributnamen als String.

Ermittlung Operationen-bezogener Information

Anmerkung: Zwar spricht die Java-Reflection-API hier konsequent von *Methoden*, bietet jedoch keine Zugriffsmöglichkeiten auf den Implementierungskörper einer Operation. Ermittelt sind daher nur die Signatur, die Sichtbarkeitsmodifier sowie der Rückgabety und, falls existent, im Ausführungsverlauf ausgelöste Ausnahmeereignisse; mithin: die *Operation*.

Die Reflection-API trennt Konstruktoren und sonstige Operationen voneinander. Daher existieren mit [getConstructor\(Class\[\]\)](#) und [getConstructors\(\)](#) bzw. [getMethod\(String, Class\[\]\)](#) und [getMethods\(\)](#) jeweils eigene Methoden zu ihrer Ermittlung.

Zurückgeliefert wird jedoch in allen Fällen genau ein, bzw. ein Array von, [Method](#)-Objekt(en).

- Modifier ([siehe Kapitel 2](#)) -- [getModifiers\(\)](#).
Der Aufruf liefert die verschiedenen Details als int-Wert, der die gesetzten Modifier Bit-codiert enthält. Zur Entschlüsselung der einzelnen Modifier sollten die Methoden der Reflection-Klasse [Modifier](#) eingesetzt werden, um auch hinsichtlich zukünftiger Java-Spracherweiterungen stabilen Code zu produzieren.
Die [Umsetzung im Beispiel](#) nutzt die Implementierung der [toString\(\)](#)-Methode der Klasse [Modifier](#), welche die in der [Java Language Specification](#) definierte kanonische Repräsentation zurückliefert.
- Übergabeparameter -- [getParameterTypes\(\)](#).
Liefert einen Array von [Class](#)-Objekten, die die formalen Parameter der Operation repräsentieren.
[Verwendung im Beispiel](#)
- Rückgabety (einer der [Primitivtypen](#), ein Array, eine beliebige zugreifbare Klasse oder eine Schnittstelle) -- [getReturnType\(\)](#).
[Verwendung im Beispiel](#)
- Auslösbare Ausnahmeereignisse -- [getExceptionTypes\(\)](#).
Liefert einen Array von [Class](#)-Objekten der deklarierten Exceptions der Operation.
[Verwendung im Beispiel](#)

```
(1)import java.lang.reflect.Constructor;
(2)import java.lang.reflect.Method;
(3)import java.lang.reflect.Modifier;
(4)import java.lang.reflect.Member;
(5)import java.lang.reflect.Field;
(6)import java.lang.ClassNotFoundException;
(7)
(8)public class ShowClass {
(9)    public static void main(String[] args) {
(10)        try {
(11)            printClass( Class.forName(args[0]) );
(12)        } catch (ClassNotFoundException cnfe) {
(13)            System.out.println("cannot find class "+args[0]);
(14)        } //catch
(15)    } //main()
(16)// -----
(17)    public static void printClass(Class c) {
(18)        Package pck = c.getPackage();
(19)        if ( pck != null )
(20)            System.out.println("package "+pck.getName()+";" );
(21)
(22)        if (c.isInterface()) {
(23)            System.out.print(Modifier.toString(c.getModifiers()) + " " + c.getName());
(24)        } else
(25)            System.out.print(Modifier.toString(c.getModifiers()) + " class " +
(26)                c.getName() +
(27)                " extends " + c.getSuperclass().getName());
(28)
(29)        Class[] interfaces = c.getInterfaces();
```



```

(30)         if ((interfaces != null) && (interfaces.length > 0)) {
(31)         if (c.isInterface())
(32)             System.out.println(" extends ");
(33)         else
(34)             System.out.print(" implements ");
(35)         for(int i = 0; i < interfaces.length; i++) {
(36)             if (i > 0)
(37)                 System.out.print(", ");
(38)             System.out.print(interfaces[i].getName());
(39)         } //for
(40)     } //if
(41)     System.out.println("\n");
(42)
(43)     Constructor[] constructors = c.getDeclaredConstructors();
(44)     if ( constructors.length > 0 ) {
(45)         System.out.println("    //Constructors");
(46)         for(int i = 0; i < constructors.length; i++)
(47)             printOperation(constructors[i]);
(48)     } //if
(49)     constructors = null;
(50)
(51)     Class[] classes = c.getDeclaredClasses();
(52)     if (classes.length > 0) {
(53)         System.out.println("    //Inner Classes");
(54)         for (int i=0; i < classes.length; i++) {
(55)             if (classes[i].isInterface() == false ) {
(56)                 printClass(classes[i] );
(57)                 System.out.print("\n");
(58)             } //if
(59)         } //for
(60)         System.out.println("    //Inner Interfaces");
(61)         for (int i=0; i < classes.length; i++) {
(62)             if (classes[i].isInterface() == true ) {
(63)                 printClass(classes[i] );
(64)                 System.out.print("\n");
(65)             } //if
(66)         } //for
(67)     } //if
(68)     classes = null;
(69)
(70)     Field[] fields = c.getDeclaredFields();
(71)     if ( fields.length > 0 ) {
(72)         System.out.println("    //Attributes");
(73)         for(int i = 0; i < fields.length; i++)
(74)             printAttribute(fields[i]);
(75)     } //if
(76)     fields = null;
(77)
(78)
(79)     Method[] operations = c.getDeclaredMethods();
(80)     if ( operations.length > 0 ) {
(81)         System.out.println("    //Operations");
(82)         for(int i = 0; i < operations.length; i++)
(83)             printOperation(operations[i]);
(84)     } //if
(85)     operations = null;
(86)
(87)     System.out.print("} //end ");
(88)     if ( c.isInterface() )
(89)         System.out.print("interface ");
(90)     else
(91)         System.out.print("class ");
(92)     System.out.print( c.getName() );
(93) } //printClass(Class)
(94)// -----
(95) public static String typename(Class t) {
(96)     String brackets = "";
(97)     while(t.isArray()) {
(98)         brackets += "[]";
(99)         t = t.getComponentType();
(100)     } //while
(101)     return t.getName() + brackets;
(102) } //typename(Class)
(103)// -----
(104) public static String modifiers(int m) {
(105)     if (m == 0)
(106)         return "";
(107)     else
(108)         return Modifier.toString(m) + " ";
(109) } //modifiers(int)
(110)// -----
(111) public static void printAttribute(Field f) {

```

```

(112)         System.out.print(" " + modifiers(f.getModifiers()) + typename(f.getType()) + " " +
f.getName() + "; //"+f.getType().getName()+" is ");
(113)
(114)         if ( (f.getType()).isInterface() )
(115)             System.out.println("an interface");
(116)         else {
(117)             if ( f.getType().isPrimitive() )
(118)                 System.out.println("a primitive type");
(119)             else {
(120)                 if ( f.getType().isArray() )
(121)                     System.out.println("an array");
(122)                 else
(123)                     System.out.println("a class");
(124)             } //else
(125)         } //else
(126)     } //printAttribute(Field)
(127)// -----
(128)     public static void printOperation(Member member) {
(129)         Class returnType=null, parameters[], exceptions[];
(130)
(131)         if (member instanceof Method) {
(132)             Method m = (Method) member;
(133)             returnType = m.getReturnType();
(134)             parameters = m.getParameterTypes();
(135)             exceptions = m.getExceptionTypes();
(136)         } else {
(137)             Constructor c = (Constructor) member;
(138)             parameters = c.getParameterTypes();
(139)             exceptions = c.getExceptionTypes();
(140)         } //else
(141)
(142)         System.out.print(" " + modifiers(member.getModifiers()) + ((returnType!=null)? typename
(returnType)+" " : "") + member.getName() + "(");
(143)         for(int i = 0; i < parameters.length; i++) {
(144)             if (i > 0)
(145)                 System.out.print(", ");
(146)
(147)             System.out.print(typename(parameters[i]));
(148)         } //for
(149)
(150)         System.out.print(")");
(151)
(152)         if (exceptions.length > 0)
(153)             System.out.print(" throws ");
(154)
(155)         for(int i = 0; i < exceptions.length; i++) {
(156)             if (i > 0)
(157)                 System.out.print(", ");
(158)
(159)             System.out.print(typename(exceptions[i]));
(160)         } //for
(161)
(162)         System.out.println(";");
(163)     } //printOperation(Member)
(164)} //class ShowClass
(165)
(166)// -----
(167)interface TestInterface {
(168)} //interface TestInterface
(169)
(170)class TestClass {
(171)     public          TestInterface i1;
(172)     private         TestClass c1;
(173)     protected      int p1;
(174)     int[]           ta1;
(175)     int[][]          tam1;
(176)
(177)     class Inner {
(178)         int i;
(179)         class InnerInner {
(180)             int j;
(181)         } //class InnerInner
(182)     } //class Inner
(183)     interface InnerInterface {
(184)     } //interface InnerInterface
(185)
(186)     private TestClass testc() {
(187)         return new TestClass() {
(188)             public void hello() {
(189)                 System.out.println("overridden test!");
(190)             } //hello()
(191)         }; //class TestClass

```

```
(192) } //test()
(193)} //class TestClass
```

Beispiel 83: Erfragen verschiedener Klassen-spezifischer Informationen durch die Reflection-API [ShowClass.java](#)

Bildschirmausgabe:

```
class TestClass extends java.lang.Object
{
    //Constructors
    TestClass();
    //Inner Classes
    class TestClass$Inner extends java.lang.Object
    {
        //Constructors
        TestClass$Inner(TestClass);
        //Inner Classes
        class TestClass$Inner$InnerInner extends java.lang.Object
        {
            //Constructors
            TestClass$Inner$InnerInner(TestClass$Inner);
            //Attributes
            int j; //int is a primitive type
            final TestClass$Inner this$1; //TestClass$Inner is a class
        } //end class TestClass$Inner$InnerInner
        //Inner Interfaces
        //Attributes
        int i; //int is a primitive type
        final TestClass this$0; //TestClass is a class
    } //end class TestClass$Inner
    //Inner Interfaces
    abstract static interface TestClass$InnerInterface
    {
    } //end interface TestClass$InnerInterface
    //Attributes
    public TestInterface i1; //TestInterface is an interface
    private TestClass c1; //TestClass is a class
    protected int p1; //int is a primitive type
    int[] ta1; //I is an array
    int[][] tam1; //[I is an array
    //Operations
    private TestClass testc();
} //end class TestClass
```

Das Ausgabe enthält, erwartungsgemäß, die implizite Superklasse `java.lang.Object`, welche durch den Übersetzer automatisch gesetzt wird, als explizite Angabe nach dem `extends`-Schlüsselwort.

Die inneren Klassen erscheinen in ihrer JVM-internen Namensgebung mit dem trennenden Dollarsymbol `$`. Auffallend ist die explizite Wiedergabe der `this`-Variable und des automatisch generierten Konstruktors mit dem Typ der äußeren Klasse als Übergabeparameter.

Für Array-Typen erfolgt die Benennung ebenfalls gemäß der JVM-internen Konvention.

Erzeugung neuer Objekte und Modifikationen bestehender mit der Reflection API

Neben den bisher vorgestellten Mechanismen zur Introspektion bestehender Strukturen kann die Reflection API auch zur Erzeugung neuer Objekte und Operation auf diesen eingesetzt werden. Hierfür stehen neben Methoden zum lesenden und schreibenden Zugriff auf Attribute auch Möglichkeiten zur Ausführung von Methoden auf den Objekten zur Verfügung.

Erzeugung von Objekten

Die Erzeugung neuer Objekte aus bestehenden Class-Ausprägungen war [bereits mit Methoden der Class-Klasse möglich](#), sofern zur Objekterzeugung der parameterlose Standardkonstruktor verwendet wird.

Zur Instanziierung neuer Objekte unter Nutzung eines bestehenden parametrisierten Konstruktors ist die Verwendung der Reflection-Klasse [Constructor](#) unabdingbar.

Der Aufruf eines parametrisierten Konstruktors vollzieht sich in drei Schritten:

1. Ermittlung des Class-Objekts für die neu zu erzeugende Instanz
2. Erzeugung eines neuen Constructor-Objekts mit der [getConstructor\(Class\[\]\)](#)-Methode der Klasse `Class`
3. Objekterzeugung durch die [newInstance\(Object\[\]\)](#)-Methode der Constructor-Klasse

```

(1)import java.lang.reflect.Constructor;
(2)import java.lang.reflect.InvocationTargetException;
(3)import java.awt.Rectangle;
(4)
(5)class SampleInstance {
(6)    public static void main(String[] args) {
(7)        Rectangle rectangle;
(8)        Class rectangleDefinition;
(9)        Class[] intArgsClass = new Class[] {int.class, int.class};
(10)        Integer height = new Integer(12);
(11)        Integer width = new Integer(34);
(12)        Object[] intArgs = new Object[] {height, width};
(13)        Constructor intArgsConstructor;
(14)
(15)        try {
(16)            rectangleDefinition = Class.forName("java.awt.Rectangle");
(17)            intArgsConstructor = rectangleDefinition.getConstructor(intArgsClass);
(18)
(19)            System.out.println ("Constructor: " + intArgsConstructor.toString());
(20)            Object object = null;
(21)
(22)            object = intArgsConstructor.newInstance(intArgs);
(23)            System.out.println ("Object: " + object.toString());
(24)            rectangle = (Rectangle) object;
(25)        } catch (ClassNotFoundException e) {
(26)            System.out.println("A ClassNotFoundException occurred!\n"+e.toString()+"\n"+e.
getMessage() );
(27)        } catch (NoSuchMethodException e) {
(28)            System.out.println("A NoSuchMethodException occurred!\n"+e.toString()+"\n"+e.
getMessage() );
(29)        } catch (InstantiationException e) {
(30)            System.out.println("An InstantiationException occurred!\n"+e.toString()+"\n"+e.
getMessage() );
(31)        } catch (IllegalAccessException e) {
(32)            System.out.println("An IllegalAccessException occurred!\n"+e.toString()+"\n"+e.
getMessage() );
(33)        } catch (IllegalArgumentException e) {
(34)            System.out.println("An IllegalArgumentException occurred!\n"+e.toString()+"\n"+e.
getMessage() );
(35)        } catch (InvocationTargetException e) {
(36)            System.out.println("An InvocationTargetException occurred!\n"+e.toString()+"\n"+e.
getMessage() );
(37)        } //catch
(38)    } //main()
(39)} //class SampleInstance

```

Beispiel 84: Erzeugung eines AWT-Rechtecks über einen parametrisierten Konstruktor mit der Reflection-API [SampleInstance.java](#)

Bildschirmausgabe:

```

Constructor: public java.awt.Rectangle(int,int)
Object: java.awt.Rectangle[x=0,y=0,width=12,height=34]

```

Das Beispiel spiegelt die drei Erstellungsschritte wieder:

Zunächst wird über den Namen (Aufruf: `forName(...)`) das zur Klasse gehörige Class-Objekt erfragt.

Die Methode `getConstructor(...)` liefert das Konstruktorenobjekt welches die als Parameter übergebene Signatur aufweist.

Instanziiert wird das neue Objekt durch die Methode `newInstance`, ausgeführt auf dem Constructor-Objekt mit den definierten Übergabeparametern.

Attributzugriff

... geschieht über Objekte der Klasse `Field`.

Für alle Primitivtypen existieren lesende Methoden, die einheitlich mit `getT(Object)` signiert sind. *T* ist einer der Primitivtypen ist. Der Rückgabetyt ist jeweils *T*.

Werte von objektartigen Attributen können mittels [get\(Object\)](#) erfragt werden; zurückgegeben wird hier eine Ausprägung von `Object`.

Symmetrisch sind die schreiben Zugriffe als `setT(Object, T)` realisiert. Mit [set\(Object, Object\)](#) steht eine Methode zur Ablage objektartiger Attribute zur Verfügung.

Das nachfolgende Beispiel zeigt die Verwendung der Lese- und Schreibmethoden.

```

(1)import java.lang.reflect.Field;
(2)
(3)public class AttributeAccess {
(4)    public static void main(String[] args) {
(5)        Test o1 = new Test();
(6)        o1.setI(42);
(7)
(8)        System.out.println( "i="+o1.getI() );
(9)
(10)       try {
(11)           Field iAttribute = (o1.getClass()).getField("i");
(12)           System.out.println( "i="+iAttribute.getInt(o1) );
(13)
(14)           iAttribute.setInt(o1, 50 );
(15)           System.out.println( "i="+iAttribute.getInt(o1) );
(16)       } catch (NoSuchFieldException e) {
(17)           System.out.println("A NoSuchFieldException occurred!\n"+e.toString()+"\n"+e.
getMessage() );
(18)           e.printStackTrace();
(19)       } catch (IllegalAccessException e) {
(20)           System.out.println("A IllegalAccessException occurred!\n"+e.toString()
+"\n"+e.getMessage() );
(21)           e.printStackTrace();
(22)       } //catch
(23)    } //main()
(24)} //class AttributeAccess
(25)
(26)class Test {
(27)    public int i;
(28)
(29)    public void setI(int i) {
(30)        this.i = i;
(31)    } //setI()
(32)
(33)    public int getI() {
(34)        return i;
(35)    } //getI()
(36)} //class Test

```

Beispiel 85: Attributzugriff mit Methoden der Reflection-Klasse Field [AttributeAccess.java](#)

BildschirmAusgabe:

```

i=42
i=42
i=50

```

Ausführen von Methoden

Die Klasse [Method](#) weist neben den bisher vorgestellten Funktionalitäten zur Inspektion auch mit der Methode [invoke](#) ([Object](#), [Object\[\]](#)) einen generischen Mechanismus zur Ausführung beliebiger Methoden auf.

Das folgende Beispiel zeigt den Aufruf der überladenen Methode hello über die invoke-Methode der Reflection-API.

```

(1)import java.lang.reflect.Method;
(2)import java.lang.reflect.InvocationTargetException;
(3)
(4)public class MethodExecution {
(5)    public static void main(String[] args) {
(6)        test o1 = new test();
(7)
(8)        try {
(9)            Method helloMethod = (o1.getClass()).getMethod("hello", null);
(10)           helloMethod.invoke(o1, null);
(11)
(12)           Class[] formalParameters = { String.class };
(13)           helloMethod = (o1.getClass()).getMethod("hello", formalParameters );
(14)           Object[] argumentList = { "stranger" };
(15)           helloMethod.invoke(o1, argumentList);
(16)       } catch (NoSuchMethodException e) {
(17)           System.out.println("A NoSuchMethodException occurred!\n"+e.toString()
+"\n"+e.getMessage() );
(18)           e.printStackTrace();
(19)       } catch (IllegalAccessException e) {
(20)           System.out.println("A NoSuchMethodException occurred!\n"+e.toString()
+"\n"+e.getMessage() );
(21)           e.printStackTrace();
(22)       } catch (InvocationTargetException e) {
(23)           System.out.println("An InvocationTargetException occurred!\n"+e.toString()

```

```

+"\\n"+e.getMessage() );
(24)         e.printStackTrace();
(25)     } //catch
(26) } //main()
(27)} //class MethodExecution
(28)
(29)class Test {
(30)    public void hello() {
(31)        System.out.println("simple hello!");
(32)    } //hello()
(33)
(34)    public void hello(String name) {
(35)        System.out.println("hello "+name);
(36)    } //hello()
(37)} //class Test

```

Beispiel 86: Ausführung zweiter Methoden durch die Reflection-API [MethodExecution.java](#)

BildschirmAusgabe:

```

simple hello!
hello stranger

```

Weitere Anwendungsbeispiele

Unter Einsatz der Reflection-API lassen sich generische Programme entwickeln, die auf beliebigen Typen operieren. Mit diesem Mechanismus lässt sich ein Verhalten vergleichbar der parametrischen Polymorphie der C++-Templates nachbilden.

```

(1)import java.lang.reflect.Array;
(2)
(3)public class GrowableArrayTest {
(4)    public static void main(String[] args) {
(5)        int[] ia = {1};
(6)        System.out.println("ia consists of "+ia.length+" elements");
(7)        ia = (int[]) arrayGrow(ia);
(8)        System.out.println("ia consists of "+ia.length+" elements");
(9)
(10)       int[] ib = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
(11)       System.out.println("ib consists of "+ib.length+" elements");
(12)       ib = (int[]) arrayGrow(ib);
(13)       System.out.println("ib consists of "+ib.length+" elements");
(14)
(15)       Person[] pa = new Person[3];
(16)       pa[0] = new Person("hans");
(17)       pa[1] = new Person("franz");
(18)       pa[2] = new Person("fritz");
(19)       System.out.println("pa consists of "+pa.length+" elements");
(20)       pa = (Person[]) arrayGrow(pa);
(21)       System.out.println("pa consists of "+pa.length+" elements");
(22)   } //main()
(23)
(24)   static Object arrayGrow(Object oldArray) {
(25)       Class arrayClass = oldArray.getClass();
(26)       if ( !arrayClass.isArray() )
(27)           return null;
(28)
(29)       Class componentType = arrayClass.getComponentType();
(30)
(31)       int oldLength = Array.getLength(oldArray);
(32)       int newLength = ( ((oldLength * 1.1) > oldLength + 1) ? (int) (oldLength * 1.1) :
(33)       (oldLength + 1) );
(34)       Object newArray = Array.newInstance(componentType, newLength );
(35)
(36)       System.arraycopy(oldArray, 0, newArray, 0, oldLength);
(37)       return newArray;
(38)   } //arrayGrow()
(39)} //class growableArrayTest
(40)
(41)class Person {
(42)    protected String name;
(43)
(44)    public Person (String name) {
(45)        this.name = name;
(46)    } //constructor
(47)} //class Person

```

Beispiel 87: Ein wachsender Array [GrowableArrayTest.java](#)

BildschirmAusgabe:

```
ia consists of 1 elements
ia consists of 2 elements
ib consists of 20 elements
ib consists of 22 elements
pa consists of 3 elements
pa consists of 4 elements
```

Das Beispiel implementiert dynamische, durch expliziten Methodenaufruf, wachsende Arrays. Nach Ermittlung des Class-Objekts (durch `getClass()`) und des Komponententyps des Arrays (`getComponentType()`) wird ein zweiter -- der neue -- Array durch die `newInstance`-Methode der Klasse `Array` erzeugt. Als Übergabeparameter wird der Komponententyp und die Länge des neuen Arrays übergeben. Die Länge des neuen Arrays entspricht der des Alten, um zehn Prozent, jedoch mindestens ein Element, erhöht. Abschließend wird der Inhalt des alten in den neu erzeugten Array kopiert. Hierzu wird die sehr performant ausgeführte native Methode `System.arraycopy` eingesetzt.

Durch die Konstruktion des Method-Objekts und der `invoke`-Methode lässt sich auch das aus C/C++ bekannte Verhalten der **Funktionszeiger** nachbilden. Das Beispiel zeigt zweimaligen Aufruf derselben Methoden `printTable`. Im Körper der Methode wird jeweils die als Parameter übergebene Methode ausgeführt. Im ersten Aufruf wird das Quadrat der Zahlen von Eins bis Zehn gebildet, im Zweiten hingegen die Wurzel gezogen.

```
(1)import java.lang.reflect.Method;
(2)import java.lang.reflect.InvocationTargetException;
(3)
(4)public class MethodPointerTest {
(5)    public static void main(String[] args) {
(6)        try {
(7)            printTable(1,10, MethodPointerTest.class.getMethod("square", new Class[]
{double.class} ));
(8)            printTable(1,10, java.lang.Math.class.getMethod("sqrt", new Class[] { double.
class} ));
(9)        } catch (NoSuchMethodException e) {
(10)            System.out.println("A NoSuchMethodException occurred!\n"+e.toString()
+"\n"+e.getMessage() );
(11)        } //catch
(12)    } //end main()
(13)
(14)    public static double square (double x) {
(15)        return (x*x);
(16)    } //square()
(17)
(18)    public static void printTable(double from, double to, Method f) {
(19)        System.out.println(f);
(20)        for (double x=from; x <= to; x++) {
(21)            System.out.print( x );
(22)            try {
(23)                Object[] args = { new Double(x) };
(24)                Double d = (Double) f.invoke(null,args);
(25)                double y = d.doubleValue();
(26)                System.out.println(" | "+y);
(27)            } catch (InvocationTargetException e) {
(28)                System.out.println("An InvocationTargetException occurred!\n"+e.
toString()+"\n"+e.getMessage() );
(29)            } catch (IllegalAccessException e) {
(30)                System.out.println("An IllegalAccessException occurred!\n"+e.
toString()+"\n"+e.getMessage() );
(31)            } //catch
(32)        } //for
(33)    } //printTable()
(34)} //class MethodPointerTest
```

Beispiel 88: Verweise auf Methoden in Java [MethodPointerTest.java](#)

BildschirmAusgabe:

```
public static double MethodPointerTest.square(double)
1.0 | 1.0
2.0 | 4.0
3.0 | 9.0
4.0 | 16.0
5.0 | 25.0
6.0 | 36.0
7.0 | 49.0
```

```

8.0 | 64.0
9.0 | 81.0
10.0 | 100.0
public static strictfp double java.lang.Math.sqrt(double)
1.0 | 1.0
2.0 | 1.4142135623730951
3.0 | 1.7320508075688772
4.0 | 2.0
5.0 | 2.23606797749979
6.0 | 2.449489742783178
7.0 | 2.6457513110645907
8.0 | 2.8284271247461903
9.0 | 3.0
10.0 | 3.1622776601683795

```

▲ 3.2.7 Abstract Windowing Toolkit (AWT)

Das AWT stellt den wohl interessantesten Teil eines (interaktiven) Java-Programmes dar -- die Benutzerinteraktion durch eine graphische Oberfläche.

Ziel der AWT-Entwicklung war es, die verschiedenen graphischen Primitiven wie Buttons, Menüs etc.

plattformunabhängig anzubieten.

Zur Realisierung des *Look and Feel* bieten sich vier verschiedene Ansätze an:

- Umsetzung ausschließlich der gemeinsamen graphischen Komponenten aller Zielplattformen.
Hierdurch entsteht automatisch ein GUI des kleinsten gemeinsamen Nenners, welches -- insbesondere wenn man kleinste Endgeräte wie PDAs und Mobiltelefone betrachtet -- praktisch nur schwerlich einsetzbar sein dürfte.
- Umsetzung absolut aller Komponenten der verschiedenen Zielplattformen auf jeder Zielplattform.
Hierdurch entsteht automatisch ein GUI das die Vereinigungsmenge aller betrachteten Plattformen abbildet. Differenzieren die betrachteten Plattformen stark in ihrer Leistungsfähigkeit, so zieht dies mitunter sehr hohen Aufwand in der späteren Umsetzung (Emulation) der einzelnen Primitive auf der realen Plattform nach sich.
- Kapselung der plattformspezifischen API durch die neue Zugriffsschicht.
Hierdurch steht dem Anwender auf der realen Plattform die dort übliche graphische Schnittstelle in Aussehen und Verhalten zur Verfügung.
Alternativ könnte auch eine vollständig neue GUI definiert werden, die dann auf allen realen Plattformen unterstützt wird. Diesen Ansatz verfolgt die Java 2 Plattform mit *SWING*.
- Bündelung und Umsetzung einer Menge *sinnvoller*, gängiger Konzepte.
Hierdurch steht auf der realen Zielplattform immer dieselbe GUI zur Verfügung, die je nach tatsächlicher Ausgestaltung der Plattform mehr oder minder aufwendig zu implementieren ist.

Das AWT bietet neben den graphischen Primitivoperationen zum Zeichnen von einfachen geometrischen Objekten wie Linien, Rechtecken, Kreisen; Fülloperationen und Methoden zur Textausgabe einen Mechanismus zur ereignisbasierten Ablaufsteuerung an, der es erlaubt auf externe Ereignisse wie Mauseingaben zu reagieren. Ferner bietet sie die weitgehend bekannten GUI-Grundelemente wie Fenster, Dialogboxen, Menüs, ... an. Zur Entwicklung komplexerer Anwendungen steht Funktionalität zur Graphikbearbeitung und Audiowiedergabe zur Verfügung.

... und SWING?

Mit der Java Version 1.1 wurde das AWT vollständig überarbeitet, insbesondere die oftmals als Achillesferse praktisch verwendbarer Applikationen empfundene Ereignisbehandlung (engl. *event handling*). Seither verwirklicht auch Java das bereits von anderen Oberflächensystemen wie NextStep oder Windows NT bekannte *Delegation Based Event Handling*. Der Terminus beschreibt die Möglichkeit der AWT externe Ereignisse (Mausclicks, Fenster-Schließen, ...) an beliebige Objekte zur Behandlung weiterzureichen.

Ebenfalls mit der JDK-Version 1.1 wurde eine zweite Bibliothek zur Oberflächenprogrammierung vorgestellt: *SWING*. Sie geht deutlich über den in der AWT realisierten Funktionsumfang hinaus, und wird daher heute überwiegend zur Implementierung professioneller Anwendungen herangezogen.

Vorwärtsreferenz: Sie wird in Kapitel [3.2.8 Swing](#) beschrieben.

Struktur des AWT

Das gesamte AWT ist im Standard-API-Paket [java.awt](#) zusammengefaßt. Es stellt die grundlegenden graphischen Primitive zur Verfügung:

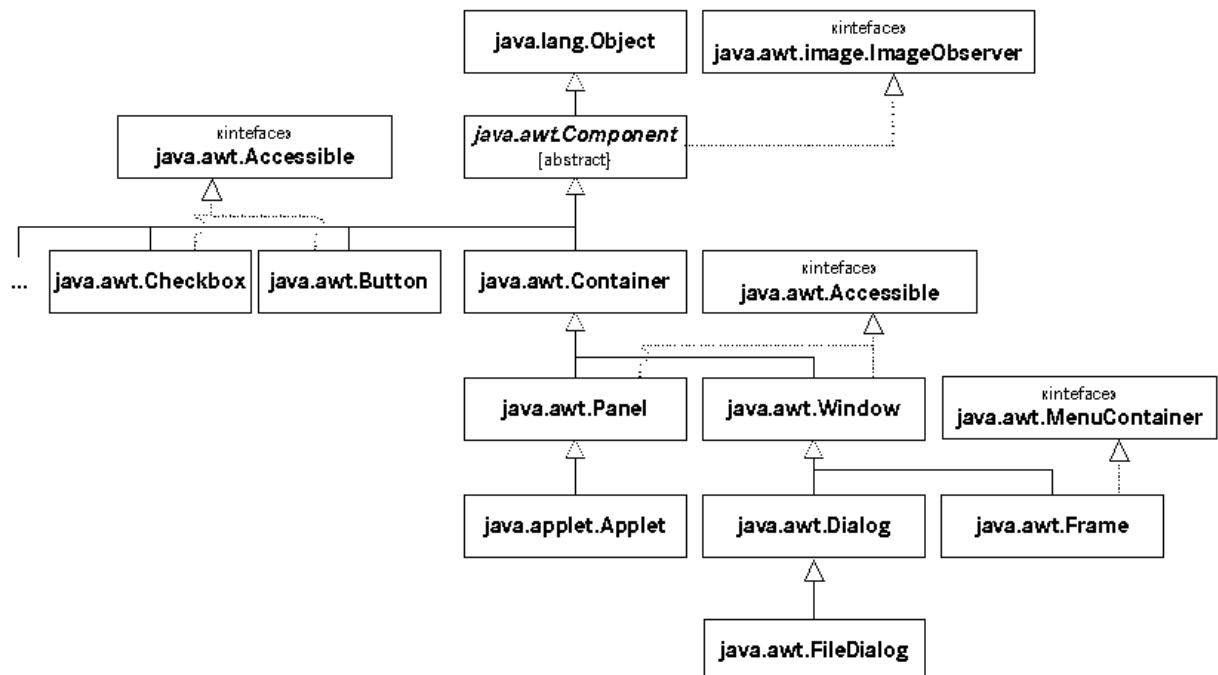
- Auswahlfelder [List](#)
enthält auswählbare Elemente.
- Beschriftungstexte [Label](#)
Einzeilige Beschriftung.
- Bildlaufleisten (scroll bars)
Verschieberegler zum Scrollen.
- [Choice](#)
Drop-down-Liste oder Optionsmenü.
- [FileDialog](#)
Dialogbox zur Auswahl von Dateien.
- leere Komponente [Canvas](#)
Kann zur Anzeige eigener Zeichnungen verwendet werden.

- Schaltflächen [Button](#)
Graphische Schaltfläche, die vom Benutzer angeklickt werden kann.
- Textbereiche (text area)
Mehrzeiliges Textfeld für Benutzereingaben und zur Textanzeige.
- Textfelder (text field)
Einzeliges Textfeld für Benutzereingaben.
- Umschaltknopf [Checkbox](#)
kann selektiert oder deselektiert werden.

Als Menü-Komponenten stehen zur Verfügung:

- [Menu](#)
Pull-down oder frei platzierbares Menü (*Abreißmenü*).
- [CheckboxMenuItem](#)
Umschaltknopf in einem Menü.
- [MenuBar](#)
... kann Pull-down-Menüs aufnehmen.
- [MenuItem](#)
Menü-Schaltfläche.
- [PopupMenu](#)
... öffnet sich automatisch bei Mauskontakt.

Die dargestellten Komponenten dienen zunächst zum Aufbau der Benutzeroberfläche (buttons, Listen, etc.) oder zur Strukturierung der Fenster (scroll bars, menus, etc.). Die dritte Kategorie bilden die Zeichenbereiche.



Ausgangspunkt jeder AWT-basierten Applikation ist die -- bereits aus der Applet-Hierarchie [bekannte](#) -- Container-Struktur.

In Containern werden die verschiedenen graphischen AWT-Primitive zusammengefaßt; Container können hierarchisch strukturiert sein, d.h. weitere Container enthalten. Der oberste Container einer Applikation bildet gleichsam die sammelnde Ordnungsstruktur. Die AWT verfügt über vier vordefinierte Containertypen:

- [Dialog](#)
Leichtgewichtiges Fenster, auch *Pop-Up-Fenster*; zumeist für Meldungen oder Eingaben benutzt. Mit der Klasse [FileDialog](#) ist ein Standard-Dialog zur Erfragung eines Dateinamens realisiert.
- [Frame](#)
Vergleichbar mit dem gleichnamigen (X)HTML-Konstrukt ermöglichen es Frames eigenständige Fenster innerhalb einer Applikation zu organisieren.
- [Panel](#)
Containertyp, der kein eigenes Fenster öffnet.
- [Window](#)
Container zur Aufnahme von Frames.

Anmerkung: Rückblickend mag es (anfänglich) verwundern, daß die Graphikwiedergabe in Applets ohne Kenntnis und Anwendung der Containerstruktur möglich war. Dies ist jedoch nur auf den ersten Blick der Fall. Zunächst stellt der Browser oder Appletviewer das Hauptfenster nebst den Menüleisten zur Verfügung -- insofern existiert ein Container zur Aufnahme des Applets bereits durch seine Ausführungsumgebung. Zusätzlich stellt das Applet selbst, durch seine Platzierung in der Vererbungshierarchie unterhalb der Klasse `java.awt.Panel`, einen eigenständigen Container dar.

Layout-Manager

Ein weiterer Ausweis der Zielsetzung plattformunabhängigen Designs des AWT ist die verwirklichte Philosophie zur Platzierung der graphischen Elemente am Bildschirm. Während plattformspezifische GUI-Bibliotheken zumeist die angebotenen Primitive mit absoluten Koordinaten im Anwendungsfenster verankern verfolgt Java hier einen

gegensätzlichen Weg.

Durch den Programmierer wird ausschließlich die Platzierung der Komponente im Verhältnis zu anderen Komponenten angegeben, die endgültige Ausrichtung am Schirm nimmt eine zusätzliche AWT-Einheit -- der *Layout-Manager* -- vor. Die Orientierung der Primitiven innerhalb einer Komponente erfolgt weiterhin durch absolute Positionierung.

Im AWT stehen folgende vordefinierte Layout-Manager zur Verfügung:

- Flow -- [FlowLayout](#)
Ordnet die enthaltenen Komponenten zentriert von links nach rechts an.
Eignet sich besonders gut zur Ausrichtung von Button-Leisten.
- Border -- [BorderLayout](#)
Orientiert Komponenten in fünf verschiedene Bereiche.
- Grid -- [GridLayout](#)
Plaziert jedes Element an einer Position innerhalb eines Quadratgitters freiwählbarer Größe.
- Card -- [CardLayout](#)
Organisiert enthaltene Container als Stapel von Karten, die jeweils einzeln angezeigt werden können.
- GridBag -- [GridBag](#)
Richtet Komponenten verschiedener Dimensionen horizontal und vertikal aus.

Anmerkungen:

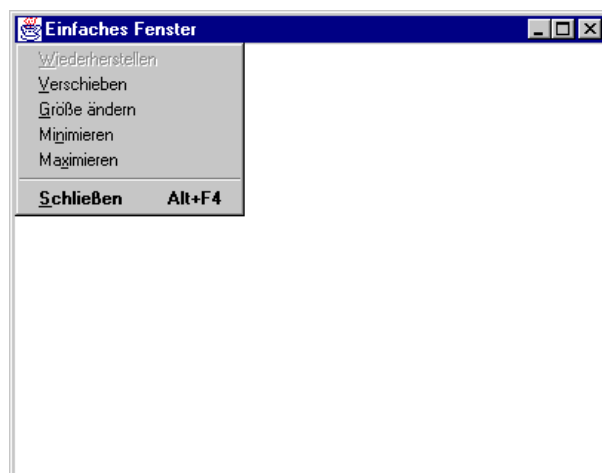
- Wird die Methode [setLayout\(LayoutManager\)](#) mit dem Übergabeparameter null aufgerufen, so wird kein Layoutmanager verwendet, sondern die Positionierung der graphischen Elemente vollständig dem Anwender überlassen.
- In realen Anwendungen wird üblicherweise massiver Gebrauch von der Möglichkeit der Schachtelung von Layoutmanagern gemacht.
Hierfür wird der Inhalt eines Frames durch mehrere Panels definiert, die unterschiedliche Layoutmanager nutzen.
- Durch Implementierung der Schnittstelle [LayoutManager](#) lassen sich eigene Layout-Manager erzeugen.

Ein erstes Fenster

```
(1)import java.awt.Frame;
(2)
(3)public class AWTE1 {
(4)    public static void main(String[] args) {
(5)        Frame wnd = new Frame("Einfaches Fenster");
(6)
(7)        wnd.setSize(400,300);
(8)        wnd.setVisible(true);
(9)    } //end main()
(10)} //class AWTE1
```

Beispiel 89: Ein einfaches Fenster mit AWT [AWTE1.java](#)

BildschirmAusgabe unter MS-Windows



Das Beispiel erzeugt einen Frame mit dem Titel *Einfaches Fenster*.

Durch die, von Component ererbte, Methode [setSize\(int, int\)](#) wird dem leeren Frame die Größe von 400 mal 300 Pixeln zugewiesen.

Zum Abschluß muß der Anzeigevorgang per [setVisible\(boolean\)](#) (ebenfalls von Component ererbt) explizit angestoßen werden.

Die Beiden Methoden `setSize` und `setVisible` greifen auf die visuellen Eigenschaften der Komponente zu, diese und weitere Methoden zu Modifikation der graphischen Erscheinung stehen auf allen Subklassen von Component zur Verfügung.

- [setSize\(Dimension\)](#) und [setSize\(int, int\)](#)
Ändert Größe der Komponente.
- [setVisible\(boolean\)](#)
Zeigt Komponente an, oder verbirgt sie.
- [setLocation\(int, int\)](#)

Verschiebt Komponente an angegeben Position.

- [`setBounds\(int, int, int, int\)`](#)
Kombination aus `setSize` und `setLocation`, ändert Lage und Größe der Komponente.
- [`requestFocus\(\)`](#)
Beantragt Eingabefokus.
- [`transferFocus\(\)`](#)
Gibt Eingabefocus an nächste Komponente weiter.
- [`setEnabled\(boolean\)`](#)
Aktiviert oder deaktiviert Komponente. Nur aktivierte Komponenten reagieren auf Eingaben und können Ereignisse verarbeiten. Vorgabegemäß sind alle Komponenten aktiviert.
- [`setFont\(Font\)`](#)
Setzt Zeichensatz der Komponente.
- [`setComponentOrientation\(ComponentOrientation\)`](#)
Setzt sprach- und kulturspezifische Orientierung der Komponente. Nach dem übergebenen Orientierungsschema werden alle Komponentenelemente und Texte ausgerichtet.
Zur Verfügung stehen derzeit `LEFT_TO_RIGHT` und `RIGHT_TO_LEFT`
- [`setCursor\(Cursor\)`](#)
Setzt Mauszeiger-Bild.
- [`setBackground\(Color\)`](#)
Setzt Hintergrundfarbe der Komponente.
- [`setForeground\(Color\)`](#)
Setzt Vordergrundfarbe der Komponente.

Auf Ausprägungen der Klasse [`Frame`](#) sind ferner verfügbar:

- [`setState\(int\)`](#)
Schaltet zwischen den Zuständen `Frame.NORMAL` (Vorgabe) und `Frame.ICONIFIED` (verkleinert) um.
- [`setIconImage\(Image\)`](#)
Definiert Icon zur Anzeige bei minimierter Darstellung.
- [`setMenuBar\(MenuBar\)`](#)
Definiert Pull-down-Menü zur Verwendung innerhalb des Frames.
- [`setResizable\(boolean\)`](#)
Erlaubt oder Verbietet Größenänderung durch Anwender.
- [`setTitle\(String\)`](#)
Setzt Titelzeile des Frames.

Bei der Ausführung fällt zunächst die Existenz der plattformüblichen Standardmenüs auf. Sie werden automatisch durch die Ausführungsumgebung zur Verfügung gestellt und mit Funktionalität versehen.

Jedoch zeigt sich, daß es nicht möglich ist das Fenster mit den üblichen Betriebssystem-spezifischen Methoden (`Close`-Button, Auswahl des entsprechenden Menüs, Tastenkombination, etc.) zu schließen. Erst der Abbruch des Prozesses der virtuellen Maschine terminiert die Applikation.

Denn anders als die üblichen Vorgabeaktionen ist das hierfür erforderliche Applikationsverhalten durch den Anwender selbst bereitzustellen.

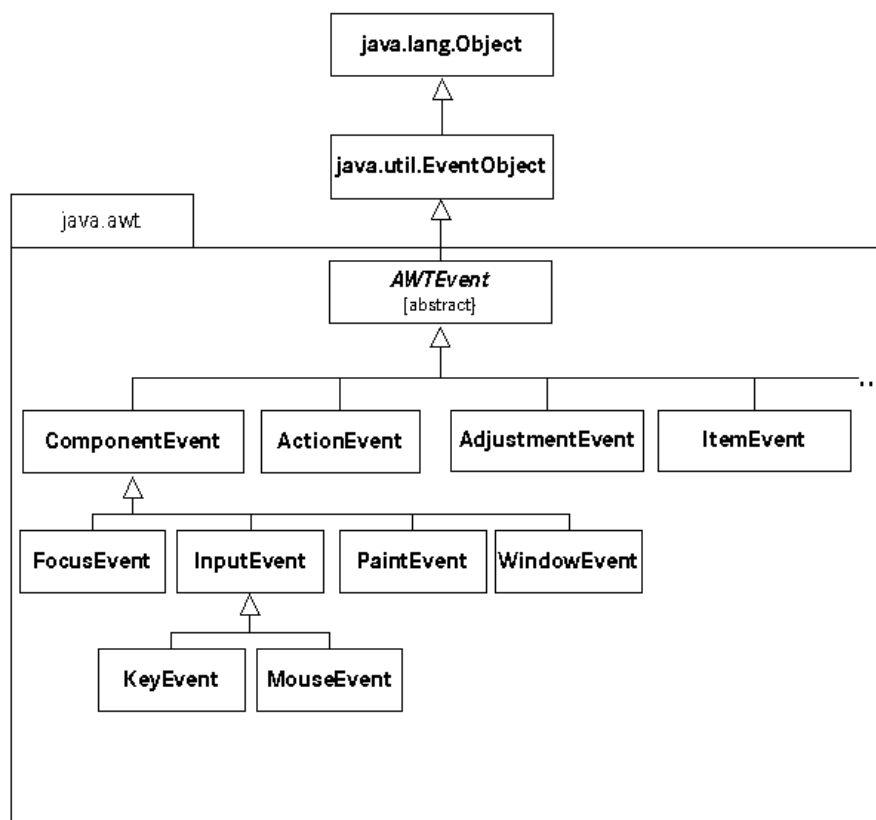
Ereignisbehandlung

Im Allgemeinen erfolgt die Kommunikation zwischen einer graphischen Oberfläche und dem ausführenden Betriebssystem in Form von Nachrichten, die zwischen diesen beiden Kommunikationspartnern ausgetauscht werden. Das Betriebssystem unterrichtet die Applikation über den Eintritt bestimmter Ereignisse (engl. *event*), hierzu zählen u.a. Mausbewegungen und -klicks, Tastaturanschläge sowie alle Fensteroperationen.

Am Nachrichtenverkehr sind prinzipiell drei Objekte des Systems beteiligt: Die Ereignisquelle, das Objekt bei dem das Ereignis eintrat (z.B. im Falle eines Mausklicks: der entsprechende Button); der Ereignisempfänger, ein Objekt das auf das eingetretene Ereignis reagiert; die Nachricht selbst, in objektorientierten System als eigenständiges Objekt realisiert, welches das ausgelöste Ereignis näher beschreibt.

Dieser Kommunikationsmechanismus ist als *Delegation Based Event Handling* bekannt, da jedes Ereignis an eine konkrete Instanz delegiert wird, die nachfolgend die Behandlung übernimmt. Es bietet zwei entscheidende Vorteile:

- Verringerung des Nachrichtenverkehrs innerhalb der Applikation, durch klare Entscheidbarkeit des Nachrichtenweges. Insbesondere unterbleibt das Senden von Nachrichten, für die kein Empfänger existiert.
- Klare Trennung zwischen GUI-Code und Applikationslogik.

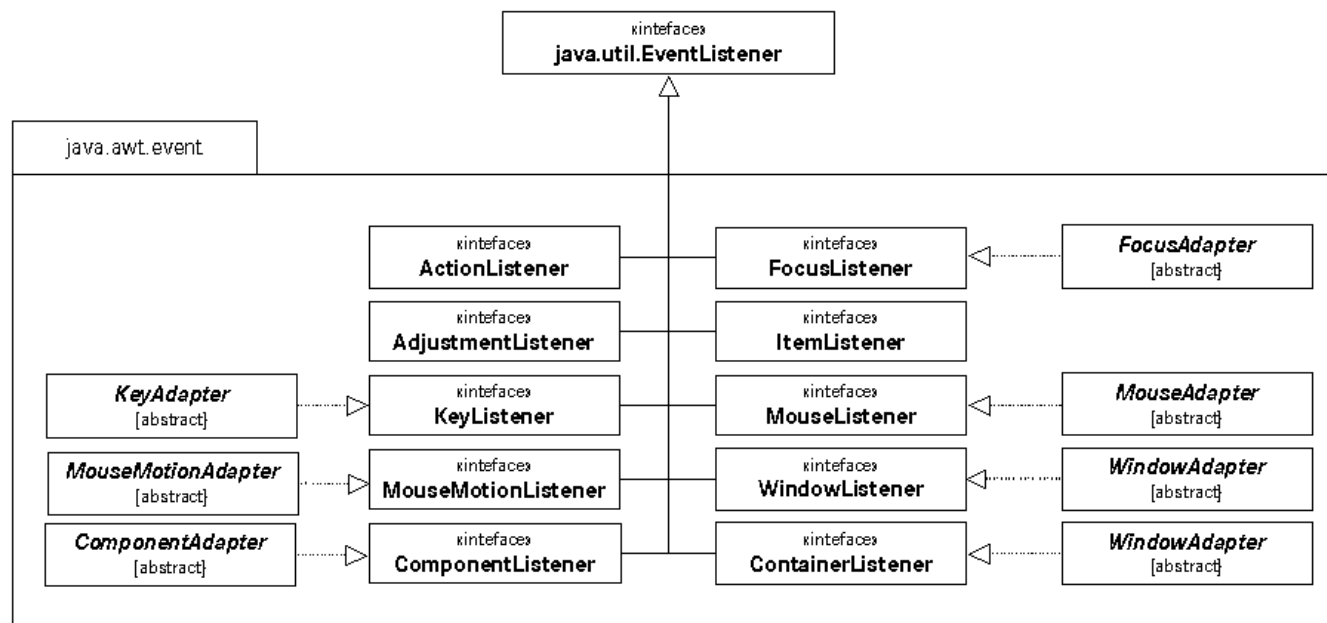


Die Abbildung zeigt eine Auswahl der verschiedenen Ereignistypen. Wie dargestellt sind alle spezifischen Ereignisse von der Klasse [EventObject](#) abgeleitet. Neben den AWT-Ereignissen existiert eine Fülle weiterer Ereignistypen, die in den verschiedenen spezifischen Paketen (wie `java.beans`, `javax.sound`, etc.) untergebracht sind.

Durch die Superklasse `EventObject` wird auch die an alle Unterklassen vererbte Methode [getSource\(\)](#) definiert, welche zu jeder `EventObject`-Ausprägung die auslösende Instanz liefert.

Unterhalb der Klasse [ComponentEvent](#) sind die *low-level*-Ereignisse eingeordnet, die auf der Ebene einer visuellen Komponente auftreten können.

Die Ereignisse [ActionEvent](#), [ItemEvent](#) und [TextEvent](#) werden als *semantische* Ereignisse bezeichnet, da sie nicht an ein konkretes visuelles Element gebunden sind.



Zur Verarbeitung der verschiedenen Nachrichtentypen muß der Empfänger eine Reihe von Methoden implementieren, die durch entsprechende Schnittstellen definiert sind. Die Abbildung stellt eine Auswahl der angebotenen Schnittstellen nebst den bereits angebotenen Implementierungen dar.

Schablone zur Reaktion auf AWT-Ereignisse

Die Reaktion auf AWT-Ereignisse wird durch sog. *event handler* übernommen; diese Objekte müssen die Benachrichtigung im Falle des Auftretens eines Ereignisses explizit anmelden.

Die Schritte im Einzelnen:

1. Definition einer Event-Handler-Klasse durch Implementierung der entsprechenden Schnittstelle
2. Registrierung des Event-Handlers

Somit kann auch das [vorangegangene Beispiel](#) entsprechend erweitert werden, um die noch ausstehende Reaktion auf das *Beenden*-Ereignis zu verwirklichen.

```
(1)import java.awt.Frame;
(2)import java.awt.Window;
(3)import java.awt.event.WindowAdapter;
(4)import java.awt.event.WindowEvent;
(5)
(6)public class AWTE2 {
(7)    public static void main(String[] args) {
(8)        Frame wnd = new Frame("Einfaches Fenster");
(9)        wnd.addWindowListener( new WindowClosingAdapter() );
(10)        wnd.setSize(400,300);
(11)        wnd.setVisible(true);
(12)    } //main()
(13)} //class AWTE2
(14)
(15)class WindowClosingAdapter extends WindowAdapter {
(16)    public void windowClosing( WindowEvent we) {
(17)        Window wnd = we.getWindow();
(18)
(19)        wnd.setVisible(false);
(20)        wnd.dispose();
(21)        System.exit(0);
(22)    } //windowClosing()
(23)} //class WindowClosingAdapter
```

Beispiel 90: Ein einfaches Fenster mit AWT [AWTE2.java](#)

Die Applikation erweitert das bisherige Beispiel um eine Adapter-Klasse, welche auf Fensterereignisse reagiert. Hierzu wird eine Ausprägung der Adapter-Klasse `WindowClosingAdapter` als Window Listener mit der Methode [addWindowListener\(WindowListener\)](#) registriert.

Die Adapter-Klasse erweitert die bestehende API-Klasse [WindowAdapter](#) und implementiert damit die Schnittstelle [WindowListener](#). Sie definiert die überschriebene Methode [windowClosing\(WindowEvent\)](#), die bei auftreten des Ereignisses automatisch zur Verarbeitung aufgerufen wird.

Nach Ermittlung des Ereignis-sendenden Fensters per [getWindow\(\)](#) wird dieses Fenster zunächst ausgeblendet (`setVisible(false)`) und im Anschluß zerstört (`dispose()`).

Danach terminiert die Applikation die virtuelle Maschine.

```
(1)import java.awt.Button;
(2)import java.awt.FlowLayout;
(3)import java.awt.Frame;
(4)import java.awt.Label;
(5)import java.awt.Window;
(6)import java.awt.event.MouseAdapter;
(7)import java.awt.event.MouseEvent;
(8)import java.awt.event.MouseEvent;
(9)import java.awt.event.MouseMotionAdapter;
(10)import java.awt.event.WindowAdapter;
(11)import java.awt.event.WindowEvent;
(12)
(13)public class AWTE3 {
(14)    private Label mousePos,
(15)                counterVal;
(16)
(17)    public static void main(String[] args) {
(18)        AWTE3 app = new AWTE3();
(19)    } //main()
(20)
(21)    public AWTE3() {
(22)        Frame wnd = new Frame("Einfaches Fenster");
(23)        mousePos = new Label("mouse pos");
(24)        counterVal = new Label("0");
(25)        Button counter = new Button("add one!");
(26)
(27)        wnd.setSize(200,100);
(28)        wnd.setLayout(new FlowLayout() );
(29)        wnd.add (mousePos);
(30)        wnd.add(counter);
(31)        wnd.add(counterVal);
(32)
(33)        wnd.addWindowListener( new WindowClosingAdapter() );
(34)        wnd.addMouseMotionListener( new MyMouseMotionAdapter(this) );
(35)        counter.addMouseListener( new CounterClicked(this) );
(36)        wnd.addMouseListener( new WindowClicked(this) );
(37)
(38)        wnd.setVisible(true);
(39)    } //constructor
(40)
(41)    public void setMousePos(String newText) {
```

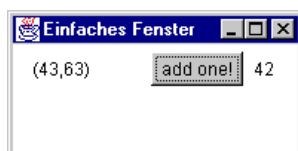
```

(42)         mousePos.setText( newText );
(43)     } //setMousePos()
(44)
(45)     public void incrementCounter() {
(46)         counterVal.setText( ""+(Integer.parseInt(counterVal.getText()+1) );
(47)     } //incrementCounter()
(48)     public void decrementCounter() {
(49)         counterVal.setText( ""+(Integer.parseInt(counterVal.getText())-1) );
(50)     } //decrementCounter()
(51)} //class AWTEX3
(52)
(53)class WindowClosingAdapter extends WindowAdapter {
(54)     public void windowClosing(WindowEvent we) {
(55)         Window wnd = we.getWindow();
(56)
(57)         wnd.setVisible(false);
(58)         wnd.dispose();
(59)         System.exit(0);
(60)     } //windowClosing()
(61)} //class WindowClosingAdapter
(62)
(63)class CounterClicked extends MouseAdapter {
(64)     private AWTEX3 app;
(65)
(66)     public CounterClicked(AWTEX3 app) {
(67)         this.app = app;
(68)     } //constructor
(69)
(70)     public void mouseClicked( MouseEvent me) {
(71)         app.incrementCounter();
(72)     } //mouseClicked()
(73)} //class CounterClicked
(74)
(75)class WindowClicked extends MouseAdapter {
(76)     private AWTEX3 app;
(77)
(78)     public WindowClicked(AWTEX3 app) {
(79)         this.app = app;
(80)     } //constructor
(81)
(82)     public void mouseClicked( MouseEvent me) {
(83)         app.decrementCounter();
(84)     } //mouseClicked()
(85)} //class WindowClicked
(86)
(87)class MyMouseMotionAdapter extends MouseMotionAdapter {
(88)     private AWTEX3 app;
(89)
(90)     public MyMouseMotionAdapter(AWTEX3 app) {
(91)         this.app = app;
(92)     } //constructor
(93)
(94)     public void mouseMoved( MouseEvent me) {
(95)         app.setMousePos( "("+me.getX()+","+me.getY()+")" );
(96)     } //mouseMoved()
(97)} //class myMouseAdapter

```

Beispiel 91: Event-Handling [AWTEX3.java](#)

Bildschirmausgabe



Die Applikation definiert, unter Verwendung des Flow-Layouts, zwei Beschriftungsfelder (labels) mousePos und counterVal sowie eine mit add one! beschriftete Schaltfläche.

Anmerkung: Die Verwendung des Layoutmanagers ist zur Verarbeitung der Maus-Events zwingend notwendig! Vor Anzeige des Applikationsfensters werden vier Ereignis-Handler registriert: Der bekannte WindowClosingListener zur Umsetzung des *Schließen*-Ereignisses, der MouseMotionListener zur Verfolgung der Mausbewegungen im Hauptfenster sowie zwei MouseListener die auf Mausereignisse -- außer Bewegungen -- auf der definierten Schaltfläche und dem Hauptfenster selbst reagieren.

Die Umsetzung des WindowClosingListeners ist gegenüber [dem vorhergehenden Beispiel](#) unverändert.

Innerhalb der Klassen counterClicked und windowClicked wird jeweils die Methode mouseClicked der Schnittstelle [MouseListener](#) überschrieben, sie wurde von der Superklasse MouseAdapter geerbt. Wird ein Mausklick auf den Button registriert, so wird eine Methode der Hauptklasse awtEx3 aufgerufen, welche die Beschriftung des Textfeldes entsprechend verändert. Beim Mausklick ist es in der vorliegenden Implementierung unentscheidend, mit welcher Maustaste dieser erfolgte. Eine nähere Analyse, beispielsweise hinsichtlich der gedrückten Taste, des aufgetretenen

Ereignisses kann über [Attribute der Klasse MouseEvent](#) erfolgen.

Die Spezialisierung der Klasse [MouseMotionAdapter](#) realisiert die Methode [mouseMoved\(MouseEvent\)](#). Sie wird jeweils bei Änderung der Mauskoordinate aufgerufen. Die beiden Koordinaten werden innerhalb des Methodenkörpers mittels der angebotenen API-Methoden extrahiert und an die Hauptklasse zur Anzeige übergeben.

Event Handling und anonyme innere Klassen

Bei näherer Betrachtung der verschiedenen Event-Handler des vorhergehenden Beispiels fällt deren immergleiche Grundstruktur auf: Zunächst das Erben von einer passenden Adapterklasse um eine oder mehrere Methoden zu überschreiben. Im konkreten Beispiel wird sogar zweimal dieselbe Adapterklasse überschrieben; jedoch mit unterschiedlichem Verhalten.

Die durch die Spezialisierung entstehenden Klassen werden in allen Fällen zur Erzeugung genau eines Objekts -- des jeweiligen Listeners -- herangezogen; eine sonstige Weiter- oder Wiederverwendung in der Applikation liegt nicht vor, und ist unter Berücksichtigung des Klassendesigns auch für die Zukunft nicht anzunehmen.

Nimmt man die beiden Randbedingungen -- *Vererbung und Überschreiben ererbter Methoden* und *keine Wiederverwendung* -- zusammen, so offenbart sich das Sprachmittel der [anonymen inneren Klassen](#) als wesentlich adäquater zur Lösung der vorliegenden Problemstellung.

Unter Nutzung dieses Mechanismus ergibt lässt sich der Code des Beispiels wie folgt modifizieren:

```
(1)import java.awt.Button;
(2)import java.awt.FlowLayout;
(3)import java.awt.Frame;
(4)import java.awt.Label;
(5)import java.awt.Window;
(6)import java.awt.event.MouseAdapter;
(7)import java.awt.event.MouseEvent;
(8)import java.awt.event.MouseMotionAdapter;
(9)import java.awt.event.WindowAdapter;
(10)import java.awt.event.WindowEvent;
(11)
(12)public class AWTEX3i {
(13)    private Label mousePos,
(14)                                counterVal;
(15)
(16)    private static AWTEX3i app;
(17)
(18)    public static void main(String[] args) {
(19)        app = new AWTEX3i();
(20)    } //main()
(21)
(22)    public AWTEX3i() {
(23)        Frame wnd = new Frame("Einfaches Fenster");
(24)        mousePos = new Label("mouse pos");
(25)        counterVal = new Label("0");
(26)        Button counter = new Button("add one!");
(27)
(28)        wnd.setSize(200,100);
(29)        wnd.setLayout(new FlowLayout() );
(30)        wnd.add (mousePos);
(31)        wnd.add(counter);
(32)        wnd.add(counterVal);
(33)
(34)        wnd.addWindowListener( new WindowAdapter() {
(35)            public void windowClosing(WindowEvent we) {
(36)                Window wnd = we.getWindow();
(37)
(38)                wnd.setVisible(false);
(39)                wnd.dispose();
(40)                System.exit(0);
(41)            } //windowClosing()
(42)        } //anonymous inner class
(43)    );
(44)
(45)    wnd.addMouseMotionListener( new MouseMotionAdapter() {
(46)        public void mouseMoved( MouseEvent me) {
(47)            app.setMousePos( "("+me.getX()+", "+me.getY()+")" );
(48)        } //mouseMoved()
(49)    } //anonymous inner class
(50)    );
(51)
(52)    counter.addMouseListener( new MouseAdapter() {
(53)        public void mouseClicked( MouseEvent me) {
(54)            app.incrementCounter();
(55)        } //mouseClicked()
(56)    } //anonymous inner class
(57)    );
(58)
(59)    wnd.addMouseListener( new MouseAdapter() {
```

```

(60)             public void mouseClicked( MouseEvent me) {
(61)                 app.decrementCounter();
(62)             } //mouseClicked()
(63)         } //anonymous inner class
(64)     };
(65)
(66)         wnd.setVisible(true);
(67)     } //constructor
(68)
(69)     public void setMousePos(String newText) {
(70)         mousePos.setText( newText );
(71)     } //setMousePos()
(72)
(73)     public void incrementCounter() {
(74)         counterVal.setText( ""+(Integer.parseInt(counterVal.getText()+1) );
(75)     } //incrementCounter()
(76)
(77)     public void decrementCounter() {
(78)         counterVal.setText( ""+(Integer.parseInt(counterVal.getText())-1) );
(79)     } //decrementCounter()
(80)} //class AWTeX3i

```

Beispiel 92: Das Beispiel AWTeX3 unter Nutzung anonymer innerer Klassen [AWTeX3i.java](#)

Der entstehende Code wird deutlich kompakter, und lokalisiert zusätzlich die Handlermethoden in den Registrierungsaufrufen.

Da anonyme innere Klassen (naturgemäß) keine Konstruktoren besitzen können wurden die Methoden entsprechend modifiziert.

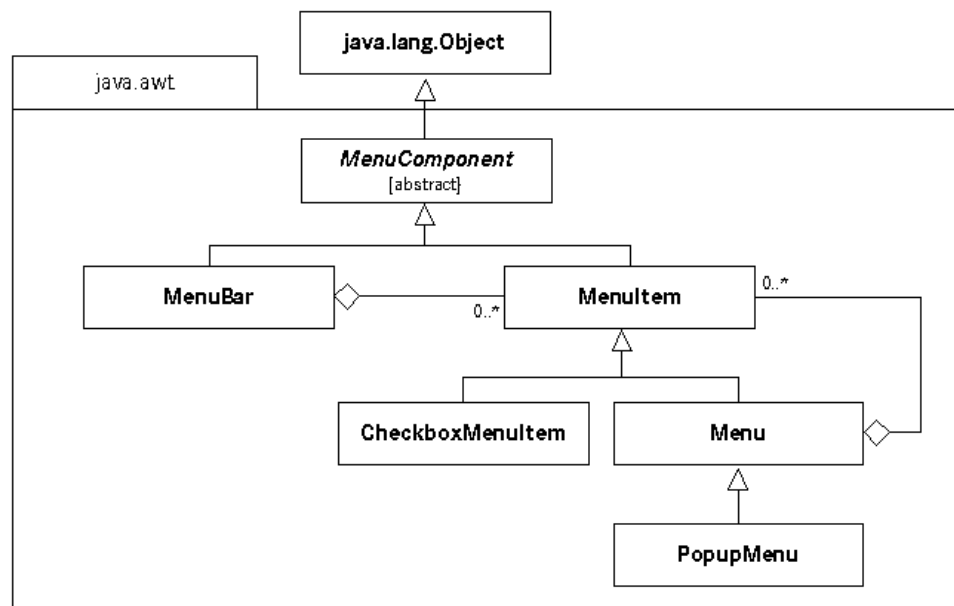
Event-Handling im Überblick

Schnittstelle	Methoden	Parameter / Zugriffsmethoden	Eventauslöser
ActionListener	actionPerformed(ActionEvent)	ActionEvent getActionCommand() getModifiers()	Button List MenuItem TextField
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)	AdjustmentEvent getAdjustable() getAdjustmentType() getValue()	Scrollbar
ItemListener	itemStateChanged(ItemEvent)	ItemEvent getItem() getItemSelectable() getStateChange()	Checkbox CheckboxMenuItem Choice List
TextListener	textValueChanged(TextEvent)	TextEvent	TextComponent
ComponentListener	componentHidden(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent) componentShown(ComponentEvent)	ComponentEvent getComponent()	Component
ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)	ContainerEvent getChild() getContainer()	Container
FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)	FocusEvent isTemporary()	Component
KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)	KeyEvent getKeyChar() getKeyCode() getKeyModifiersText(int) getKeyText() isActionKey()	Component
MouseListener	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)	MouseEvent getClickCount() getPoint() getX() getY() isPopupTrigger()	Component

MouseEvent	mouseDragged(MouseEvent)	getClickCount()	Component
	mouseMoved(MouseEvent)	getPoint()	
		getX()	
		getY()	
		isPopupTrigger()	
WindowListener	windowActivated(WindowEvent)	WindowEvent	Window
	windowClosed(WindowEvent)	getWindow()	
	windowClosing(WindowEvent)		
	windowDeactivated(WindowEvent)		
	windowDeiconified(WindowEvent)		
	windowIconified(WindowEvent)		
	windowOpened(WindowEvent)		

Menüs

Das AWT bietet die auch von anderen graphischen Benutzerschnittstellen bekannten Menüprimitive und -Grundfunktionen an. Darunter verschachtelte Menüs (Submenüs), Short-Cuts, frei platzierbare Kontextmenüs, etc. Die Abbildung zeigt die verschiedenen Primitive und ihre Beziehungen zueinander.



Die abstrakte Klasse [MenuComponent](#) bildet die Wurzel der Menüklassenhierarchie.

Ein [MenuBar](#) stellt eine Zusammenfassung von Menüs dar. Objekte dieser Klasse bilden den Ausgangspunkt einer Menüstruktur.

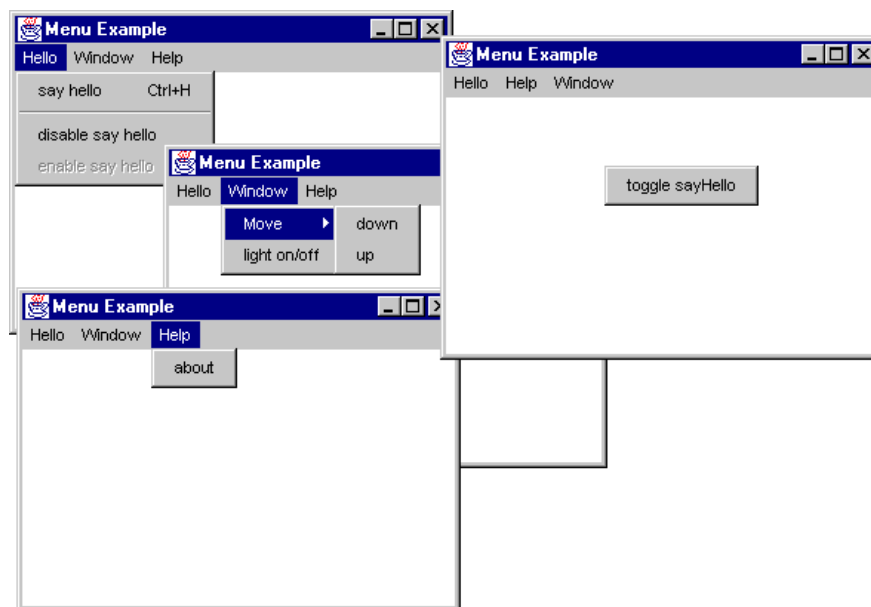
Innerhalb eines `MenuBar`s können beliebig viele [MenuItem](#)-Ausprägungen organisiert werden. Ein `MenuItem` ist ein beliebiger Eintrag eines Menüs. Hierbei kann es sich um einen anklickbaren Menüeintrag oder ein weiteres Menü handeln, welches sich bei Mausberührung öffnet.

Üblicherweise werden konkrete Menüeinträge in Menüs zusammengefaßt. Dem Menü selbst entspricht die AWT-Klasse [Menu](#). Jedes `Menu`-Objekt kann aus weiteren `MenuItem`s bestehen. Mithin kann ein Menü Submenüs oder direkte Menüeinträge in beliebiger Reihenfolge enthalten.

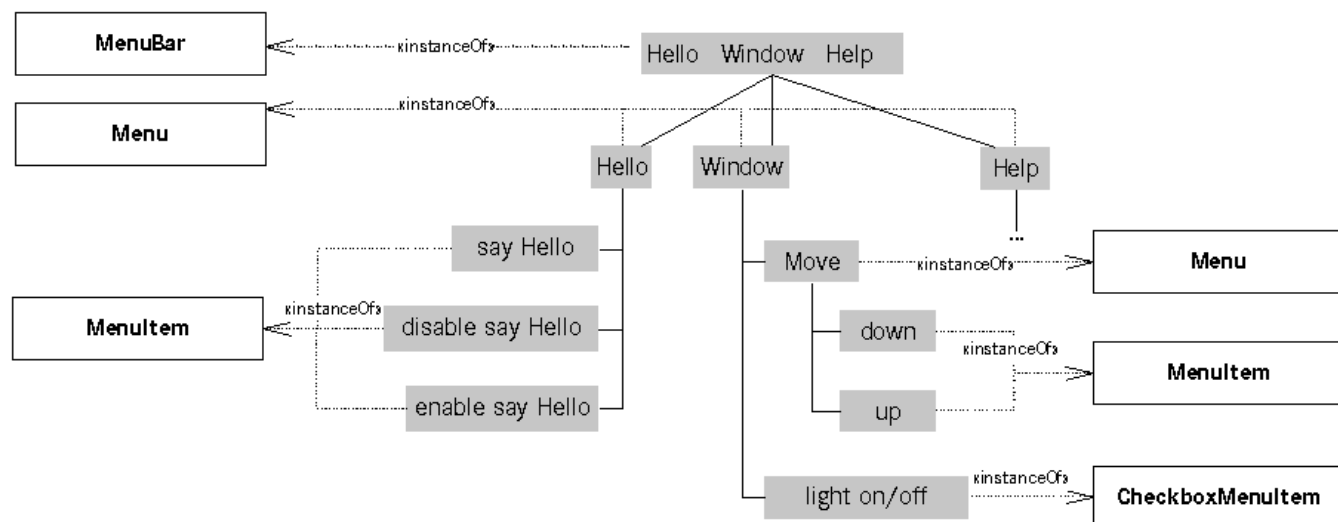
Als Spezialisierung der durch die Klasse `MenuItem` definierten Menüeinträge stehen umschaltbare Menüpunkte durch die Klasse [CheckboxMenuItem](#) zur Verfügung. Gegenüber den herkömmlichen Menüeinträgen verfügt dieser Typ über ein zusätzliches -- in der visuellen Darstellung plattformabhängig variierendes -- Symbol welches den Eintragszustand „aktiviert“ oder „deaktiviert“ anzeigt.

Menüleisten-unabhängige Kontextmenüs werden durch die Klasse [PopupMenu](#) realisiert. Sie sind frei platzierbar, verhalten sich jedoch ansonsten wie die bekannten Menüs.

Neue (klickbare) Menüeinträge werden als Objekte der Klasse [MenuItem](#) erzeugt. Die dem Konstruktor übergebene Zeichenkette legt den späteren angezeigten Eintrag im Menü fest. ([Verwendung im Beispiel](#)).



Die Abbildung stellt einen Ausschnitt der Menüstruktur der Beispielapplikation dar:



Erzeugung von Menüstrukturen

Ausgangspunkt der Menüerzeugung ist immer ein Objekt der Klasse `MenuBar`. Durch den parameterlosen Konstruktor wird eine leere Menüleiste erzeugt, die als Container zur Aufnahme der weiteren Menükomponenten dient.

([Verwendung im Beispiel](#)).

Die einzelnen Menüs werden durch Objekte der entsprechenden Klasse `Menu` repräsentiert. Der in der Menüleiste aufgeführte Menüname wird als Zeichenkettenparameter (Klasse `java.lang.String`) übergeben. ([Verwendung im Beispiel](#).)

Durch die Methode `add(MenuItem)` werden Menüeinträge oder ganze Menüs -- allgemein: Subklassen von `MenuItem` -- in eine bestehende Menüleiste eingehängt. ([Verwendung im Beispiel](#)).

Zusätzlich kann dem Konstruktor eine Instanz der Klasse `MenuShortcut` übergeben werden. Hierdurch wird die in GUIs übliche direkte Anwahlmöglichkeit einzelner Menüeinträge über Tastenkürzel realisiert. ([Verwendung im Beispiel](#)).

Zur Erzeugung von `CheckboxMenuItems`, Menüeinträgen die ihren zweiwertigen Zustand automatisch darstellen, muß ein Objekt der Klasse `CheckboxMenuItem` erzeugt und in ein bestehendes Menü eingehängt werden. ([Verwendung im Beispiel](#)).

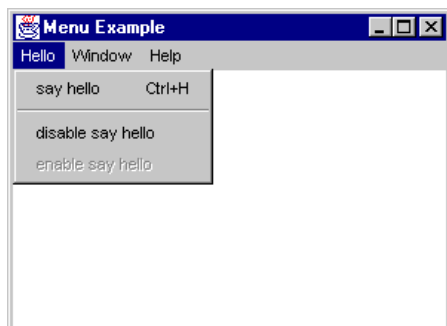
Die Variante der Pop-up Menüs wird durch die gleichnamige Klasse `PopupMenu` erzeugt. Im Unterschied zu den herkömmlichen Menüs wird dieser Menütyp nicht in der Menüleiste, sondern dem beherbergenden Fenster direkt, durch die dortige `add`-Methode, eingehängt. ([Verwendung im Beispiel](#)).

Die **Ereignisbehandlung** für Menüstrukturen folgt dem in der Übersicht [Event-Handling im Überblick](#) gegebenen Schema.

Für jeden Menüeintrag vom Typ `MenuItem` muß ein Objekt vom Typ `ActionListener` über die Methode `addActionListener(ActionListener)` der Klasse `MenuItem` registriert werden. Dieser muß die Methode `actionPerformed(ActionEvent)` überschreiben. Dort findet sich der Code, welcher bei Eintreten des Ereignisses (Mausclick auf Menüeintrag) abgearbeitet wird. ([Verwendung im Beispiel](#)).

Menüpunkte der Klasse `CheckboxMenuItem` werden durch Objekte die die Schnittstelle `ItemListener` implementieren überwacht. Tritt die Zustandsänderung ein, so wird die Methode `itemStateChanged(ItemEvent)` aufgerufen. ([Verwendung im Beispiel](#)).

```
(1)import java.awt.Button; (2)import java.awt.CheckboxMenuItem; (3)import java.awt.Color; (4)import java.awt.Dialog; (5)
import java.awt.FlowLayout; (6)import java.awt.Frame; (7)import java.awt.Label; (8)import java.awt.Menu; (9)import java.awt.
MenuBar; (10)import java.awt.MenuItem; (11)import java.awt.MenuShortcut; (12)import java.awt.Point; (13)import java.awt.
PopupMenu; (14)import java.awt.Window; (15)import java.awt.event.ActionEvent; (16)import java.awt.event.ActionListener; (17)
import java.awt.event.InputEvent; (18)import java.awt.event.ItemEvent; (19)import java.awt.event.ItemListener; (20)import
java.awt.event.KeyEvent; (21)import java.awt.event.MouseAdapter; (22)import java.awt.event.MouseEvent; (23)import java.awt.
event.WindowAdapter; (24)import java.awt.event.WindowEvent; (25) (26)public class AWTEx4 { (27) private static AWTEx4 app;
(28) private Label lbl_hello; (29) private Frame wnd; (30) private MenuItem mi_sayHello, mi_disable, mi_enable, mi_toggle,
mi_up, mi_down; (31) private CheckboxMenuItem mi_light; (32) private PopupMenu myPopupMenu; (33) (34) class
ToggleSayHelloState implements ActionListener { (35) public void actionPerformed(ActionEvent ae) { (36) mi_sayHello.
setEnabled( !mi_sayHello.isEnabled() ); (37) mi_enable.setEnabled( !mi_sayHello.isEnabled() ); (38) mi_disable.setEnabled
( mi_sayHello.isEnabled() ); (39) } //actionPerformed() (40) } //class ToggleSayHelloSate (41) (42) public static void main
(String[] args) { (43) app = new AWTEx4(); (44) } //main() (45) (46) public AWTEx4() { (47) wnd = new Frame("Menu
Example"); (48) lbl_hello = new Label("Hello!"); (49) wnd.add(lbl_hello); (50) (51) wnd.setSize(300,200); (52) wnd.setLayout
(new FlowLayout() ); (53) (54) MenuBar myMenuBar = new MenuBar(); (55) Menu m_hello = new Menu("Hello"); (56) myMenuBar.add
(m_hello); (57) mi_sayHello = new MenuItem("say hello", new MenuShortcut (KeyEvent.VK_H)); (58) m_hello.add( mi_sayHello );
(59) mi_disable = new MenuItem("disable say hello"); (60) m_hello.addSeparator(); (61) mi_enable = new MenuItem("enable say
hello"); (62) m_hello.add( mi_disable ); (63) m_hello.add( mi_enable ); (64) (65) Menu m_windowAction = new Menu("Window");
(66) Menu m_windowActionMove = new Menu("Move"); (67) mi_light = new CheckboxMenuItem("light on/off"); (68) mi_down = new
MenuItem("down"); (69) mi_up = new MenuItem ("up"); (70) (71) m_windowActionMove.add(mi_down); (72) m_windowActionMove.add
(mi_up); (73) m_windowAction.add(m_windowActionMove); (74) m_windowAction.add(mi_light); (75) myMenuBar.add
(m_windowAction); (76) (77) //pop-up menu (78) myPopupMenu = new PopupMenu(); (79) MenuItem mi_toggle = new MenuItem
("toggle sayHello"); (80) myPopupMenu.add( mi_toggle ); (81) wnd.add( myPopupMenu ); (82) (83) Menu m_hlp = new Menu
("Help"); (84) m_hlp.add(new MenuItem("about")); (85) (86) myMenuBar.setHelpMenu(m_hlp); (87) (88) wnd.setMenuBar
(myMenuBar); (89) (90) mi_up.addActionListener( new ActionListener() { (91) public void actionPerformed(ActionEvent ae)
{ (92) Point pnt = wnd.getLocation(); (93) (94) for (int x=pnt.x; x >= 0 ; x--) (95) wnd.setLocation(x, pnt.y); (96) for
(int y=pnt.y; y >= 0; y--) (97) wnd.setLocation(0, y); (98) } //actionPerformed() (99) } //anonymous inner class (100) );
(101) (102) mi_down.addActionListener( new ActionListener() { (103) public void actionPerformed(ActionEvent ae) { (104)
final Dialog dlg_notImplemented = new Dialog(wnd,true); (105) Label lbl = new Label("function not implemented, yet!");
(106) dlg_notImplemented.setTitle("Sorry, I'm afraid I can't to that ..."); (107) dlg_notImplemented.setLayout( new
FlowLayout() ); (108) dlg_notImplemented.setResizable(false); (109) dlg_notImplemented.add(lbl); (110) dlg_notImplemented.
setSize(250,80); (111) (112) Button btn_ok = new Button("ok"); (113) btn_ok.addActionListener( new ActionListener() { (114)
public void actionPerformed(ActionEvent ae) { (115) dlg_notImplemented.setVisible(false); (116) dlg_notImplemented.dispose
(); (117) } //actionPerformed() (118) } //anonymous inner class (119) ); (120) (121) dlg_notImplemented.add(btn_ok); (122)
(123) dlg_notImplemented.show(); (124) } //actionPerformed() (125) } //anonymous inner class (126) ); (127) (128) mi_light.
addItemListener( new ItemListener() { (129) public void itemStateChanged(ItemEvent ie) { (130) if (mi_light.getState())
(131) wnd.setBackground( new Color(33,33,33) ); (132) else (133) wnd.setBackground( new Color(255,255,255) ); (134) } //
actionPerformed() (135) } //anonymous inner class (136) ); (137) (138) mi_sayHello.addActionListener( new ActionListener()
{ (139) public void actionPerformed(ActionEvent ae) { (140) lbl_hello.setVisible(true); (141) try { (142) Thread.sleep
(1500); (143) } catch (InterruptedException e) { (144) //ignore it (145) } //catch (146) lbl_hello.setVisible(false);
(147) } //actionPerformed() (148) } //anonymous inner class (149) ); (150) (151) ActionListener toggler = new
ToggleSayHelloState(); (152) mi_disable.addActionListener( toggler ); (153) mi_enable.addActionListener( toggler ); (154)
mi_toggle.addActionListener( toggler ); (155) (156) wnd.addWindowListener( new WindowAdapter() { (157) public void
windowClosing(WindowEvent we) { (158) Window wnd = we.getWindow(); (159) (160) wnd.setVisible(false); (161) wnd.dispose();
(162) System.exit(0); (163) } //windowClosing() (164) } //anonymous inner class (165) ); (166) (167) wnd.addMouseListener
( new MouseAdapter() { (168) public void mouseClicked( MouseEvent me) { (169) if ( (me.getModifiers() & InputEvent.
BUTTON1_MASK) == 0 ) (170) myPopupMenu.show( me.getComponent(), me.getX(), me.getY() ); (171) } //mouseClicked() (172) } //
anonymous inner class (173) ); (174) (175) wnd.setVisible(true); (176) lbl_hello.setVisible(false); (177) mi_enable.
setEnabled(false); (178) } //main() (179) } //class AWTEx4
```

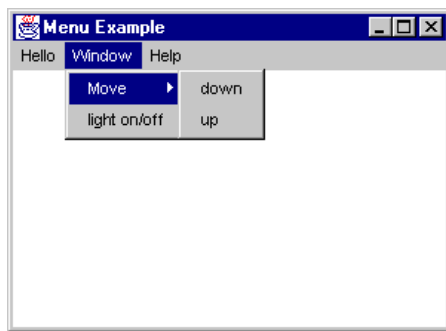
Beispiel 93: Beispiel einer Menüstruktur [AWTEx4.java](#)

Das Menü *Hello* der Beispielapplikation besteht aus drei Menüeinträgen -- *say hello*, *enable say hello* und *disable say hello*. Für den ersten Menüpunkt ist das Tastaturkürzel CTRL+H definiert ([Definition im Code](#)). Bereits zum Startzeitpunkt der Applikation ist der Menüpunkt *enable say hello* deaktiviert, da der entsprechende Menüpunkt vorgabegemäß aktiviert ist. Die Instanzenmethode [setEnabled](#) der Klasse `MenuItem` erlaubt den Wechsel zwischen den beiden Zuständen „aktiviert“, mit normaler visueller Darstellung des Menüpunktes, und „deaktiviert“, mit entsprechender *aus-gegrauter* Darstellung. Standardmäßig sind alle Menüeinträge nach ihrer Erzeugung aktiviert, daher muß die gewünschte Deaktivierung explizit erfolgen. ([Stelle im Code](#)).

Wird der Menüpunkt *disable say hello* angewählt, so zieht dies die Deaktivierung des Menüpunktes *say hello* und gleichzeitige (Re-)Aktivierung von *enable say hello* nach sich.

Die Selektierung des Menüpunktes *enable say hello* vollführt hingegen die duale Operation dazu, Aktivierung der Menüpunkte *say hello* und *disable say hello*. Aus diesem Grunde teilen sich beide Menüpunkte dieselbe Ereignisbehandlungs-Routine. Als Konsequenz dieser Forderung wird die Initialisierung der beiden `ActionListener` mit demselben Objekt notwendig ([Codestelle](#))-- daher kann die Umsetzung an dieser Stelle nicht mehr als anonyme innere Klasse realisiert werden, sondern erfolgt als „normale“ innere Klasse ([Codestelle](#)).

Die Anwahl des Menüpunktes *say hello* zeigt für eineinhalb Sekunden den Schriftzug *Hello!* am Bildschirm an. ([Ereignisbehandlungsroutine](#)).

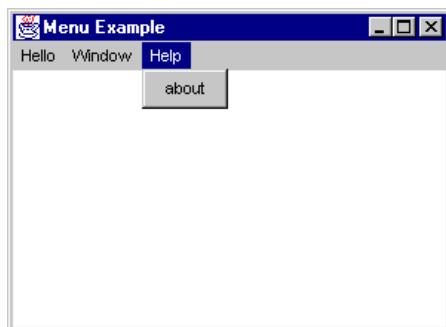


Das zweite Menü der Menüleiste, mit *Window* betitelt, enthält ein Untermenü *Move* und eine Ausprägung von `CheckboxMenuItem`.

Die Ereignisbehandlung für Submenüeinträge erfolgt analog der auf höheren Ebenen; durch Definition des entsprechenden `ActionListener`-Instanz. ([Implementierung der Ereignisbehandlung des Menüpunktes *up*](#)). Bei Auswahl dieses Menüpunkts wird das Fenster, durch mehrmalige Änderung der Bildschirmkoordinaten in die linke obere Bildschirmcke verschoben.

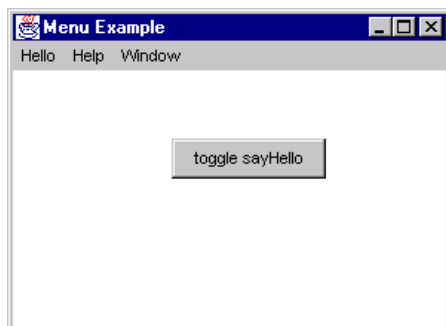
Der Eintragstyp `CheckboxMenuItem` erfordert eine Behandlungsroutine vom Typ `ItemListener`. Die Anzeige des Zustands wird durch die AWT automatisch vorgenommen, und bedarf keiner Anwenderinteraktion.

([Behandlungsroutine des Menüeintrages *light on/off*](#)). Der Menüpunkt ändert den Bildschirmhintergrund je nach Zustand von hell nach dunkel oder umgekehrt.



Das dritte dargestellte Menü wurde als Standard-Hilfe-Menü deklariert. Existiert bereits ein Menü dieser Eigenschaft, so wird es durch das neu definierte ersetzt.

Die Aufnahme in die Menüleiste erfolgt nicht durch die bekannte `add`-Methode, sondern über den separaten Aufruf `setHelpMenu(Menu)` ([Codestelle](#)).



Zusätzlich definiert die Applikation ein Kontextmenü welches die Funktionalität der Menüeinträge *enable say hello* und *disable say hello* vereinigt. Auch es nutzt die Ereignisbehandlungsroutine der beiden genannten Menüeinträge mit. ([Codestelle](#)).

Die Aktivierung und Anzeige am Punkt der Aktivierung muß durch den Anwender selbst realisiert werden. Hierfür ist ein entsprechender `MouseListener` umzusetzen. Die Methode `show(Component, int, int)` erlaubt hierfür die Übergabe von Koordinaten, an denen das Kontextmenü aufgeklappt wird.

Textfelder

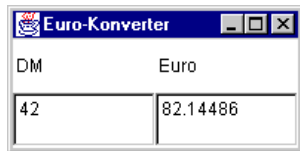
Editierbare Textfelder für Benutzereingaben stellt die Klasse `TextField` bereit.

Zur Reaktion auf Tastenanschläge kann -- wie in [Übersicht](#) dargestellt -- jedes Objekt genutzt werden, das die von `Component` ererbte Schnittstelle `KeyListener` implementiert.

```
(1)import java.awt.Frame; (2)import java.awt.GridLayout; (3)import java.awt.Label; (4)import java.awt.TextField; (5)import
java.awt.Window; (6)import java.awt.event.KeyAdapter; (7)import java.awt.event.KeyEvent; (8)import java.awt.event.
WindowAdapter; (9)import java.awt.event.WindowEvent; (10) (11)public class AWTEX5 { (12) private static AWTEX5 app; (13)
private TextField tf_dm, tf_eur; (14) (15) public static void main(String[] args) { (16) app = new AWTEX5(); (17) } //main
() (18) (19) public AWTEX5() { (20) Frame wnd = new Frame("Euro-Konverter"); (21) (22) wnd.setSize(200,100); (23) wnd.
setLayout(new GridLayout(2,2) ); (24) (25) Label lbl_dm = new Label ("DM"); (26) Label lbl_eur = new Label ("Euro"); (27)
tf_dm = new TextField("1", 7); (28) tf_eur = new TextField("1.95583", 7 ); (29) (30) wnd.add(lbl_dm); (31) wnd.add
(lbl_eur); (32) wnd.add(tf_dm); (33) wnd.add(tf_eur); (34) wnd.show(); (35) (36) (37) tf_dm.addKeyListener( new KeyAdapter
() { (38) public void keyReleased(KeyEvent e) { (39) tf_eur.setText( ""+Double.parseDouble(tf_dm.getText())*1.95583 );
(40) } //keyPressed() (41) } //anonymous inner class (42) ); (43) (44) tf_eur.addKeyListener( new KeyAdapter() { (45)
public void keyReleased(KeyEvent e) { (46) tf_dm.setText(""+Double.parseDouble(tf_eur.getText())/1.95583 ); (47) } //
keyPressed() (48) } //anonymous inner class (49) ); (50) (51) wnd.addWindowListener( new WindowAdapter() { (52) public void
windowClosing(WindowEvent we) { (53) Window wnd = we.getWindow(); (54) (55) wnd.setVisible(false); (56) wnd.dispose(); (57)
System.exit(0); (58) } //windowClosing() (59) }//anonymous inner class (60) ); (61) } //constructor (62)} //end class AWTEX5
```

Beispiel 94: Ein einfacher Euro-Umrechner [AWTEX5.java](#)

Bildschirmausgabe:



Die Applikation nutzt das [GridLayout](#) zur Ausrichtung der vier visuellen Komponenten. Hierfür wird der entsprechende Layoutmanager dahingehend parametrisiert, daß er ein quadratisches Layout mit jeweils zwei horizontalen und vertikalen Einträgen erlaubt. ([Codestelle](#)).

Die Beschriftungen der Spalten sind als Labels realisiert.

Zum Erzeugungszeitpunkt wird den Text-Feldern ein Startwert, und die horizontale Ausdehnung von sieben Zeichen zugewiesen. ([Codestelle](#)).

In den (anonymen inneren) Klassen zur Ereignisbehandlung, wird die Methode [keyTyped\(KeyEvent\)](#) überschrieben. Sie wird *nach* Abschluß des Tastendrucks aufgerufen.

In der Implementierung dieser Methode wird die tatsächliche Umrechnung vorgenommen, und daß Umrechnungsergebnis im jeweils anderen Feld ausgegeben. ([Codestelle](#)).

Hinweis: Eine Behandlung des möglicherweise generierten `NumberFormatException`-Ausnahmeereignisses findet aus Übersichtlichkeitsgründen nicht statt.

Dialoge

Ausprägungen der Klasse [Dialog](#) stellen eine Möglichkeit zur Realisierung einfacher Ein- und Ausgabefenster dar.

Diese Art Fenster eignet sich besonders für kurze Anfragen, oder Meldungen, an den Anwender.

[Beispiel `awtEx4`](#) enthält eine solche Nachrichtenbox. Dialogfenster können, im Gegensatz zu den bekannten Applikationsfenstern, *modal* sein. Dies bedeutet, daß ein solches Fenster nicht durch den Anwender in den Hintergrund versetzt werden kann; es erzwingt eine Reaktion -- zumeist in Form der Bestätigung einer Meldung o.ä. -- bevor die Applikation fortfährt.

▲ 3.2.8 Swing

Mit dem JDK v1.1 wurde als zusätzliche API zur Erstellung von graphischen Oberflächen [Swing](#) vorgestellt. Bis zur aktuellen Version 1.3 ist diese API weiterhin Bestandteil des [Extension Paketes](#), und steht daher nicht auf allen Plattformen zur Verfügung. Des weiteren können sich noch zukünftige Änderungen an den derzeit publizierten APIs ergeben, so daß bestehender Code überarbeitet werden muß.

Folgende Überlegungen führten zur Entwicklung von Swing:

- Durch die Übernahme des plattformspezifischen look-and-feels der AWT-Applikationen entsteht teilweise großer Portierungsaufwand zur Realisierung nicht-nativ verfügbarer Primitive.
- Jede Änderung an den realen GUIs der Zielsysteme läßt Modifikationen an der AWT-Implementierung notwendig werden.
Dies konterkariert die angestrebte Plattformunabhängigkeit auf der Ebene der angebotenen GUIs zunehmend.
- Aufwendige und komplexe Benutzeroberflächen sind mit dem AWT u. U. sehr schwer zu realisieren.

Daher wurde mit den *Java Foundation Classes*, deren Bestandteil Swing ist, versucht eine echte Alternative zum bestehenden -- und weiter angebotenen -- AWT zu schaffen. Dabei erhöht Swing massiv die Anzahl der angebotenen visuellen Komponenten, und fügt einige sehr mächtige Primitive wie beispielsweise zur Darstellung von Bäumen hinzu.

Anders als die AWT setzt Swing nicht auf den Möglichkeiten des zugrundeliegenden GUI-Systems auf, sondern stellt selbst die gesamte Verwaltung und Verarbeitung der angebotenen Primitiven zur Verfügung. Als Folge dieses Ansatzes realisiert Swing ein eigenes look-and-feel, das auf allen unterstützten Plattformen unverändert präsentiert wird. Mit diesem -- *Metal* genannten -- Layout entsteht erstmals ein typisches Aussehen nativer Java-Applikationen. Technisch ist Swing mit den seit JDK v1.1 angebotenen *leichtgewichtigen Komponenten*, den Java Beans, realisiert. Die Realisierung graphischer Primitivoperationen erfolgt daher nicht mehr durch Operationen des zugrundeliegenden

GUI-Systems, sondern durch die angebotenen Swing-Komponenten selbst.

Dies führt dazu, daß die gesamte Swing-API ohne native Methoden -- vollständig in Java codiert -- realisiert werden konnte. Hieraus ergeben sich weitere Vorteile in der Anwendung, insbesondere in der Fehlersuche.

Jedoch führt die verfolgte *pure Java*-Implementierung auch zu Problemen. So erhöht sich durch den de-facto emulierenden Ansatz der Speicherplatzbedarf, da nicht mehr auf die evtl. durch die Plattform angebotenen Routinen zurückgegriffen wird. Flankierend sinkt die Ausführungsgeschwindigkeit durch die gestiegene Menge an auszuführendem Java-Code.

Anmerkung:

Aus rein praktischen Erwägungen sprechen derzeit noch zwei weitere Punkte gegen den breiten Swing-Einsatz: zunächst der Reifegrad und die Entwicklungsstabilität. Als Bestandteil des Java Extension Paketes ist Swing explizit als experimenteller Bestandteil der derzeit verfügbaren Java-API gekennzeichnet. In der verbreiteten Implementierung finden sich noch kleinere und größere Fehler und Ungereimtheiten, die gegen die Verwendung in Produktivapplikationen sprechen. Wie bei allen anderen Bestandteilen des *javax*-Paketes auch, behält sich SUN explizit das Recht vor Schnittstellen und Funktionalität ohne Ankündigung zu verändern oder aus dem Angebot herauszunehmen. Daher verbietet sich, unter Berücksichtigung einer zukünftigen Pflege des entstehenden Applikationscodes, die Nutzung dieser Klassen schon fast.

Ergänzend sei noch erwähnt, daß seitens der aktuell verfügbaren Web-Browser noch keine Swing-Unterstützung umgesetzt ist. Dieses Manko läßt sich zwar durch die zusätzliche manuelle Installation des [Java-Plugins](#) beheben, verlangt dem Anwender jedoch einen zusätzlichen Installationsvorgang ab.

Swing verwirklicht durchgängig das von T. Renskaug initiierte *Model-View-Controller*-Konzept. Dessen hervorstechendstes Kennzeichen die Separierung des Codes in drei verschiedene Verwendungskategorien ist:

- **Model** -- Enthält die Daten und den Zustand des Dialogelements
- **View** -- Übernimmt die graphische Darstellung der visuellen Komponente
- **Controller** -- Verbindet *Model* und *View*; empfängt Ereignisse und leitet entsprechende Maßnahmen an *Model* und *View* ein

Die gesamte Verarbeitungslogik ist dabei in der *Model*-Klasse zentralisiert. Zu jedem *Model* können dabei gleichzeitig verschiedene *View*-Klassen existieren.

Swing reduziert diesen Mechanismus auf zwei verschiedenen Klassen-Typen. Hierbei wird die *View*- und die *Controller*-Komponente zu einer Einheit verschmolzen. Das entstehende Design wird *Model-Delegate-Prinzip* genannt.

```
(1)import java.awt.Window;
(2)import java.awt.event.ActionEvent;
(3)import java.awt.event.ActionListener;
(4)import java.awt.event.WindowAdapter;
(5)import java.awt.event.WindowEvent;
(6)import javax.swing.JButton;
(7)import javax.swing.JFrame;
(8)import javax.swing.JPanel;
(9)import javax.swing.SwingUtilities;
(10)import javax.swing.UIManager;
(11)import javax.swing.UnsupportedLookAndFeelException;
(12)
(13)public class SwingEx1 extends JFrame implements ActionListener {
(14)    public SwingEx1() {
(15)        super("first swing application");
(16)        JPanel myPanel = new JPanel();
(17)        JButton btn_win = new JButton("Windows");
(18)        JButton btn_motif = new JButton("Motif");
(19)        JButton btn_metal = new JButton("Metal");
(20)
(21)        btn_win.setToolTipText("switch to Windows look-and-feel");
(22)        btn_motif.setToolTipText("switch to Motif look-and-feel");
(23)        btn_metal.setToolTipText("switch to Metal look-and-feel");
(24)
(25)        myPanel.add( btn_win );
(26)        myPanel.add( btn_motif );
(27)        myPanel.add( btn_metal );
(28)
(29)        btn_win.addActionListener(this);
(30)        btn_motif.addActionListener(this);
(31)        btn_metal.addActionListener(this);
(32)
(33)        getContentPane().add("South", myPanel);
(34)
(35)        addWindowListener( new WindowAdapter() {
(36)            public void windowClosing(WindowEvent we) {
(37)                Window wnd = we.getWindow();
(38)
(39)                wnd.setVisible(false);
(40)                wnd.dispose();
(41)                System.exit(0);
(42)            } //windowClosing()
(43)        } //anonymous inner class
(44)    };
(45) } //constructor
(46)
```

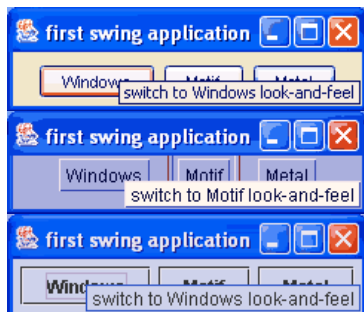
```

(47)     public void actionPerformed(ActionEvent ae) {
(48)         String cmd = ae.getActionCommand(),
(49)             plaf="";
(50)
(51)         if ( cmd.equals("Metal") ) {
(52)             plaf = "javax.swing.plaf.metal.MetalLookAndFeel";
(53)         } else {
(54)             if ( cmd.equals("Motif") ) {
(55)                 plaf = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
(56)             } else {
(57)                 if( cmd.equals("Windows") ) {
(58)                     plaf = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel";
(59)                 } //else
(60)             } //else
(61)         } //else
(62)
(63)         try {
(64)             UIManager.setLookAndFeel(plaf);
(65)             SwingUtilities.updateComponentTreeUI(this);
(66)         } catch (UnsupportedLookAndFeelException e) {
(67)             System.err.println(e.toString());
(68)         } catch (ClassNotFoundException e) {
(69)             System.err.println(e.toString());
(70)         } catch (InstantiationException e) {
(71)             System.err.println(e.toString());
(72)         } catch (IllegalAccessException e) {
(73)             System.err.println(e.toString());
(74)         } //catch
(75)     } //actionPerformed()
(76)
(77)     public static void main(String[] args) {
(78)         SwingEx1 wnd = new SwingEx1();
(79)         wnd.setSize(300,200);
(80)         wnd.pack();
(81)         wnd.setVisible(true);
(82)     } //end main()
(83)} //class SwingEx1

```

Beispiel 95: Eine erstes Swing-Applikation [SwingEx1.java](#)

Bildschirmausgabe:



Diese Swing-Applikation definiert ein einfaches Fenster mit drei Buttons.

Deutlichstes Kennzeichen der Nutzung von Swing ist das Präfix *J* vor den bekannten AWT-Komponentennamen. So erweitert die Applikation das Swing-Analogon der AWT-Klasse `Frame`, jetzt unter dem Namen `JFrame`.

Die Schaltflächen sind entsprechend Ausprägungen von `JButton`. Mit der Methode `setToolTipText(String)` wird die rein Swing-interne Möglichkeit zur Definition eines Hilfstexts genutzt, der bei Überstreichen einer Komponente mit der Maus automatisch angezeigt wird. Die Implementierung greift nicht auf Mechanismen des zugrundeliegenden GUI-Systems zurück, sondern ist vollständig innerhalb von Swing realisiert.

Die Ereignisbearbeitung ist identisch zur AWT v1.1 realisiert, und kann daher nahezu unverändert übernommen werden.

Als Behandlungsroutine der drei Schaltflächen ist der Wechsel des Bildschirmlayouts zur Laufzeit realisiert. Nach Aktivierung der entsprechenden Schaltfläche wird das entsprechende, innerhalb der Swing-API vordefinierte Layouts referenziert, und durch die statische Methode `setLookAndFeel(String)` geladen. Nach der Aktualisierung der Bildschirmdarstellung präsentiert sich die gesamte Applikation im neuen Layout.

Service provided by [Mario Jeckle](#)

Generated: 2004-06-08T19:30:35+01:00

► [Feedback](#) ► [SiteMap](#)

► [This page's original location: http://www.jeckle.de/vorlesung/java/script.html](#)

► [RDF description for this page](#)



[Home](#) | [Products & services](#) | [Support & downloads](#) | [My account](#)

Select a country

Servers > Mainframe servers >



zSeries Application Assist Processor(zAAP)

Delivering a specialized z/OS Java execution environment

Today's reality

In the e-business On Demand era, business requirements change more frequently than ever. To stay current and competitive, businesses are developing new strategic web-based applications. Java adoption continues to accelerate as an open programming model but, these applications typically require more IT resources than traditional applications due to levels of abstraction, code generation, and reuse. Unfortunately, IT budgets are not keeping pace with these needs, forcing customers to seek more cost effective and productive ways of deploying new Java technology-based applications.

An exciting new optional feature

The new IBM  zSeries® Application Assist Processor (zAAP), available on the IBM eServer zSeries 990 (z990) and zSeries 890 (z890) servers, is an attractively priced specialized processing unit that provides a strategic z/OS Java™ execution environment for customers who desire the powerful integration advantages and traditional Qualities of Service of the zSeries platform.

(Unless otherwise noted, references to z/OS with regard to zAAP will also apply to z/OS.e.)

zAAPs may enable customers to:

- Simplify and reduce server infrastructures by integrating e-business Java Web applications next to mission critical data for high performance, reliability, availability and security.
- Maximize the value of their zSeries investment through increased system productivity by reducing the demands and capacity requirements on general purpose processors which may then be available for reallocation to other zSeries workloads.
- Lower the overall cost of computing for WebSphere Application Server and other Java technology-based applications; through hardware, software and maintenance savings.

When configured with general purpose processors within logical partitions running z/OS (or z/OS.e), zAAPs may help increase general purpose processor productivity and may contribute to lowering the overall cost of computing for z/OS Java technology-based applications. zAAPs are designed to operate asynchronously with the general processors to execute Java programming under control of the IBM Java Virtual Machine (JVM). This can help reduce the demands and capacity requirements on general purpose processors which may then be available for reallocation to other zSeries workloads.

The IBM JVM processing cycles can be executed on the configured zAAPs with no anticipated modifications to the Java application(s). Execution of the JVM processing cycles on a zAAP is a function of the IBM Software Developer's Kit (SDK) for z/OS, Java 2 Technology Edition V1.4, z/OS (or z/OS.e) 1.6, and the Processor Resource/Systems Manager™ (PR/SM™).

The amount of general purpose processor savings will vary based on the amount of Java application code executed by zAAP(s). This is dependent upon the amount of Java cycles used by the relevant application(s) and on the zAAP execution mode selected by the customer.

Execution of the Java applications on zAAPs, within the same z/OS SMP LPAR as their associated database subsystems, can also help simplify the server infrastructures and improve operational efficiencies. For example, use of zAAPs could reduce the number of TCP/IP programming stacks, firewalls, and physical interconnections (and their associated processing latencies) that might otherwise be required when the application servers and their database servers are deployed on separate physical server platforms.

Conceptually, zAAPs are very similar to a System Assist Processor (SAP); they cannot execute an Initial Program Load and can only assist the general purpose Central Processors (CPs) for the execution of Java programming under control of the IBM JVM. However, unlike standard CPs, ICFs and IFLs, zAAPs can do nothing on their own. For this reason, IBM does not impose software charges on zAAP capacity.

Additional IBM software charges apply only when additional general purpose CP capacity is used.

Customers are encouraged to contact their specific ISVs/USVs directly to determine if their charges will be affected.

Special Note: For customers planning to run WebSphere Application Server V5 under z/OS on z890 or z990 server, they may be eligible to take advantage of some of the benefits that the zAAP provides for a limited time on their z890 or z990 prior to migrating to z/OS (or z/OS.e) 1.6. They are encouraged to contact their local IBM or authorized business partner hardware sales specialist for more information.

For more information on z990 and zAAP, please refer to the z990 Latest Enhancements [Announcement Letter \(104-118\)](#).

For more information on the new z890 and zAAP, please refer to the z890 [Announcement Letter \(104-117\)](#).

[About IBM](#) | [Privacy](#) | [Terms of use](#) | [Contact](#)

Learn more

- [zAAP Home](#)
- [Getting started](#)
- [Pricing](#)
- [Support](#)
- [FAQ](#)

We're here to help



Easy ways to get the answers you need

- [Request a quote \(U.S.\)](#)
- [Request a quote \(Canada\)](#)

Announcing



Software



Operating systems



Feature information



Resources

- [What they're saying about zAAP](#)
- [Press releases](#)



News

Download / FTP

Documentation

Ports

Anonymous

CVS

ViewCVS /

CVSWeb

Mailing Lists

Mailing List

Archives

Bugs

Testing

Security

Compatibility

Performance

Projects

Screenshots

Hall of Fame

Hall of Shame

Links

Credits

Kaffe.org News

For announcements, see the [kaffe-announce mailing list archives](#).

February 18, 2004

Kaffe 1.1.4 has been released. This is the next in a series of "development" releases. [See the announcement](#).

[Kaffe Weekly News Archive](#)

Latest Release

Download the source code here:

Latest Development Release
(February 18, 2004)

[kaffe-1.1.4.tar.gz](#)

Latest Production Release (July 2, 2002)

[kaffe-1.0.7.tar.gz](#)

Kaffe is already distributed in package form with many Linux and *BSD distributions, including [Red Hat](#), [Mandrake](#), [SuSE](#), [Debian](#), [Gentoo](#), [Conectiva](#), [PLD](#), [Ark Linux](#), [FreeBSD](#), [NetBSD](#), [OpenBSD](#), and many others.

Kaffe can also be downloaded anonymously on [Freenet](#) at the following Freesite:

SSK@uTjAWzcMP-
CO10nI5dqlu7T%

What is Kaffe?

- Kaffe is a clean room implementation of the Java virtual machine, plus the associated class libraries needed to provide a Java runtime environment. The Kaffe virtual machine is free software, licensed under the terms of the [GNU General Public License](#).
- Kaffe is a great choice as a base for virtual machine education and/or research, or if you need a virtual machine as an integral component of an open source or free software Java distribution.

What is Kaffe not?

- Kaffe is not an officially licensed version of the Java virtual machine. In fact, it contains no Sun source code at all, and was developed without even looking at the Sun source code. It is legal -- but Sun controls the Java trademark, and has never endorsed Kaffe, so technically, **Kaffe is not Java**.
- Kaffe is constantly under development, and lacks compatibility in many ways with the current releases of Java. It lacks many key features of a full Java virtual machine



7eOgsPAgM/
jimpick/5//

implementation - including security related features such as a complete bytecode verifier essential for running untrusted code.

Legal:

Kaffe.org is a an independent, free software community project.

The Kaffe virtual machine is a clean room implementation of the Java Virtual Machine from Sun Microsystems, Inc.

The Kaffe virtual machine is free software, licensed under the terms of the GNU General Public License.

Java is a Trademark of Sun Microsystems, Inc.

All copyrights are held by their respective owners, please read and respect the individual licenses on the software distributed from this site.

Thanks:

To [Berkeley Signal Inc.](#) for donating the server hardware for Kaffe.org.

To [CommunityColo.net](#) for hosting the server. Please [donate](#) and help them to support this, and other non-profit causes. Learn about how to [sponsor](#) Kaffe.org for a month.

[Jim Pick](#)

- Kaffe is not the best Java virtual machine for developing Java applications, as it lacks much in the way of documentation, compatibility, debugging/profiling support, etc. If you are learning Java, or are looking for a complete Java development environment, you will probably be best served by using a "real" Java development environment (such as the JDK) licensed from Sun. Check out our [links page](#) for more information.
- Kaffe is not the only free software Java project. There are other worthy VM and class library implementations to consider. Please check out some of the other projects on our [links page](#).



Jikes™ Research Virtual Machine (RVM)

[Jikes RVM Home Page](#) | [Jikes RVM Project](#) | [IBM PL/SE Research](#) | [IBM Research](#) | [IBM® Home](#)

Navigation

Project Information

[Q & A overview](#)

[Project roles](#)

[How can I help?](#)

[Current users](#)

[Acknowledgements](#)

Resources

[Publications \(All\)](#)

[Presentations](#)

[Teaching resources](#)

Getting Jikes RVM

[News](#)

[Downloads](#)

[Mailing lists](#)

[Problems, bugs,
or good ideas?](#)

Developer Information

[How can I help?](#)

[Contributing to
Jikes RVM](#)

[Regression Tests](#)

[User's Guide](#)

[Jikes RVM API](#)

[Browse CVS](#)

[Watch CVS commits](#)

[Active CVS areas](#)

[Developer Login](#)

Jikes RVM (Research Virtual Machine) provides the research community with a flexible [open](#) testbed to prototype virtual machine technologies and experiment with a large variety of design alternatives. The system is licensed under the [CPL](#), which has been approved by the [OSI](#) (Open Source Initiative) as a fully certified open source license. The initial Jikes RVM infrastructure was independently developed as part of the Jalapeño research project at the [IBM T.J. Watson Research Center](#).

Jikes RVM runs on Linux®/IA-32, AIX™/PowerPC™, OS X/PowerPC, and Linux/PowerPC platforms and advances the state-of-the-art of virtual machine technologies for dynamic compilation, adaptive optimization, garbage collection, thread scheduling, and synchronization. A distinguishing characteristic of Jikes RVM is that it is implemented in the Java™ programming language and is self-hosted i.e., its Java code runs on itself without requiring a second virtual machine. Most other virtual machines for the Java platform are written in native code (typically, C or C++). A Java implementation provides ease of portability, and a seamless integration of virtual machine and application resources such as objects, threads, and operating-system interfaces.

- Jikes RVM (v2.2.1 and later) is based entirely on the class libraries produced by the [GNU Classpath](#) project.
- Jikes RVM (v2.2.1 and later) can run a significant subset of the [Eclipse](#) IDE. Details are in the [user's guide](#).
- Jikes RVM (v2.3.2 and later) can be built with completely open source software. Details are in the [user's guide](#).

Many [researchers](#) have found that Jikes RVM provides a useful vehicle for research on the frontiers of virtual machine technologies (several dozen [publications](#)), as well as [teaching courses](#).

Upcoming Events

- [PLDI'04 Tutorial](#) *Dynamic Compilation and Adaptive Optimization in Virtual Machines*, Stephen Fink, David Grove, and Michael Hind, Tuesday, June 8, 2004.
- [ISMM'04 Tutorial](#) *JMTk: The Java Memory Tool-kit*, Steve Blackburn and Perry Cheng, Saturday, October 23, 2004.

Information on the [Jikes source-to-bytecode](#) compiler.

www.ibm.com/developerworks/oss/jikesrvm

For Jikes RVM web site feedback contact [Michael Hind](#)

java.lang

Class System

[java.lang.Object](#)└─ [java.lang.System](#)

```
public final class System
```

extends [Object](#)

The `System` class contains several useful class fields and methods. It cannot be instantiated.

Among the facilities provided by the `System` class are standard input, standard output, and error output streams; access to externally defined properties and environment variables; a means of loading files and libraries; and a utility method for quickly copying a portion of an array.

Since:

JDK1.0

Field Summary

static PrintStream	err The "standard" error output stream.
static InputStream	in The "standard" input stream.
static PrintStream	out The "standard" output stream.

Method Summary

static void	arraycopy (Object src, int srcPos, Object dest, int destPos, int length) Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.
static String	clearProperty (String key) Removes the system property indicated by the specified key.
static long	currentTimeMillis () Returns the current time in milliseconds.
static void	exit (int status) Terminates the currently running Java Virtual Machine.
static void	gc () Runs the garbage collector.
static Map < String , String >	getenv () Returns an unmodifiable string map view of the current system environment.
static String	getenv (String name) Gets the value of the specified environment variable.
static Properties	getProperties () Determines the current system properties.
static String	getProperty (String key) Gets the system property indicated by the specified key.
static String	getProperty (String key, String def) Gets the system property indicated by the specified key.
static SecurityManager	getSecurityManager () Gets the system security interface.
static int	identityHashCode (Object x) Returns the same hash code for the given object as would be returned by the default method hashCode(), whether or not the given object's class overrides hashCode().
static Channel	inheritedChannel () Returns the channel inherited from the entity that created this Java virtual machine.
static void	load (String filename) Loads a code file with the specified filename from the local file system as a dynamic library.

static void	loadLibrary (String libname) Loads the system library specified by the libname argument.
static String	mapLibraryName (String libname) Maps a library name into a platform-specific string representing a native library.
static long	nanoTime () Returns the current value of the most precise available system timer, in nanoseconds.
static void	runFinalization () Runs the finalization methods of any objects pending finalization.
static void	runFinalizersOnExit (boolean value) Deprecated. <i>This method is inherently unsafe. It may result in finalizers being called on live objects while other threads are concurrently manipulating those objects, resulting in erratic behavior or deadlock.</i>
static void	setErr (PrintStream err) Reassigns the "standard" error output stream.
static void	setIn (InputStream in) Reassigns the "standard" input stream.
static void	setOut (PrintStream out) Reassigns the "standard" output stream.
static void	setProperty (Properties props) Sets the system properties to the Properties argument.
static String	setProperty (String key, String value) Sets the system property indicated by the specified key.
static void	setSecurityManager (SecurityManager s) Sets the System security.

Methods inherited from class [java.lang.Object](#)

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

Field Detail

in

```
public static final InputStream in
```

The "standard" input stream. This stream is already open and ready to supply input data. Typically this stream corresponds to keyboard input or another input source specified by the host environment or user.

out

```
public static final PrintStream out
```

The "standard" output stream. This stream is already open and ready to accept output data. Typically this stream corresponds to display output or another output destination specified by the host environment or user.

For simple stand-alone Java applications, a typical way to write a line of output data is:

```
System.out.println(data)
```

See the `println` methods in class `PrintStream`.

See Also:

[PrintStream.println\(\)](#), [PrintStream.println\(boolean\)](#),
[PrintStream.println\(char\)](#), [PrintStream.println\(char\[\]\)](#),
[PrintStream.println\(double\)](#), [PrintStream.println\(float\)](#),
[PrintStream.println\(int\)](#), [PrintStream.println\(long\)](#),
[PrintStream.println\(java.lang.Object\)](#), [PrintStream.println](#)
[\(java.lang.String\)](#)

err

```
public static final PrintStream err
```

The "standard" error output stream. This stream is already open and ready to accept output data.

Typically this stream corresponds to display output or another output destination specified by

the host environment or user. By convention, this output stream is used to display error messages or other information that should come to the immediate attention of a user even if the principal output stream, the value of the variable `out`, has been redirected to a file or other destination that is typically not continuously monitored.

Method Detail

setIn

```
public static void setIn(InputStream in)
```

Reassigns the "standard" input stream.

First, if there is a security manager, its `checkPermission` method is called with a `RuntimePermission("setIO")` permission to see if it's ok to reassign the "standard" input stream.

Parameters:

`in` - the new standard input stream.

Throws:

[SecurityException](#) - if a security manager exists and its `checkPermission` method doesn't allow reassigning of the standard input stream.

Since:

JDK1.1

See Also:

[SecurityManager.checkPermission\(java.security.Permission, RuntimePermission\)](#)

setOut

```
public static void setOut(PrintStream out)
```

Reassigns the "standard" output stream.

First, if there is a security manager, its `checkPermission` method is called with a `RuntimePermission("setIO")` permission to see if it's ok to reassign the "standard" output stream.

Parameters:

`out` - the new standard output stream

Throws:

[SecurityException](#) - if a security manager exists and its `checkPermission` method doesn't allow reassigning of the standard output stream.

Since:

JDK1.1

See Also:

[SecurityManager.checkPermission\(java.security.Permission\)](#),
[RuntimePermission](#)

setErr

```
public static void setErr(PrintStream err)
```

Reassigns the "standard" error output stream.

First, if there is a security manager, its `checkPermission` method is called with a `RuntimePermission("setIO")` permission to see if it's ok to reassign the "standard" error output stream.

Parameters:

`err` - the new standard error output stream.

Throws:

[SecurityException](#) - if a security manager exists and its `checkPermission` method doesn't allow reassigning of the standard error output stream.

Since:

JDK1.1

See Also:

[SecurityManager.checkPermission\(java.security.Permission\)](#),
[RuntimePermission](#)

inheritedChannel

```
public static Channel inheritedChannel()  

                                throws IOException
```

Returns the channel inherited from the entity that created this Java virtual machine.

This method returns the channel obtained by invoking the [inheritedChannel](#) method of the system-wide default [SelectorProvider](#) object.

In addition to the network-oriented channels described in [inheritedChannel](#), this method may return other kinds of channels in the future.

Returns:

The inherited channel, if any, otherwise `null`.

Throws:

[IOException](#) - If an I/O error occurs

[SecurityException](#) - If a security manager is present and it does not permit access to the channel.

Since:

1.5

setSecurityManager

```
public static void setSecurityManager(SecurityManager s)
```

Sets the System security.

If there is a security manager already installed, this method first calls the security manager's `checkPermission` method with a `RuntimePermission ("setSecurityManager")` permission to ensure it's ok to replace the existing security manager. This may result in throwing a `SecurityException`.

Otherwise, the argument is established as the current security manager. If the argument is `null` and no security manager has been established, then no action is taken and the method simply returns.

Parameters:

s - the security manager.

Throws:

[SecurityException](#) - if the security manager has already been set and its `checkPermission` method doesn't allow it to be replaced.

See Also:

[getSecurityManager\(\)](#), [SecurityManager.checkPermission\(java.security.Permission\)](#), [RuntimePermission](#)

getSecurityManager

```
public static SecurityManager getSecurityManager()
```

Gets the system security interface.

Returns:

if a security manager has already been established for the current application, then that security manager is returned; otherwise, `null` is returned.

See Also:

[setSecurityManager\(java.lang.SecurityManager\)](#)

currentTimeMillis

```
public static long currentTimeMillis()
```

Returns the current time in milliseconds. Note that while the unit of time of the return value is a millisecond, the granularity of the value depends on the underlying operating system and may be larger. For example, many operating systems measure time in units of tens of milliseconds.

See the description of the class `Date` for a discussion of slight discrepancies that may arise between "computer time" and coordinated universal time (UTC).

Returns:

the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.

See Also:

[Date](#)

nanoTime

```
public static long nanoTime()
```

Returns the current value of the most precise available system timer, in nanoseconds.

This method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time. The value returned represents nanoseconds since some fixed but arbitrary time (perhaps in the future, so values may be negative). This method provides nanosecond precision, but not necessarily nanosecond accuracy. No guarantees are made about how frequently values change. Differences in successive calls that span greater than approximately 292 years (2^{63} nanoseconds) will not accurately compute elapsed time due to numerical overflow.

For example, to measure how long some code takes to execute:

```
long startTime = System.nanoTime();
// ... the code being measured ...
long estimatedTime = System.nanoTime() - startTime;
```

Returns:

The current value of the system timer, in nanoseconds.

Since:

1.5

arraycopy

```
public static void arraycopy(Object src,
                             int srcPos,
                             Object dest,
                             int destPos,
                             int length)
```

Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array. A subsequence of array components are copied from the source array referenced by `src` to the destination array referenced by `dest`. The number of components copied is equal to the `length` argument. The components at positions `srcPos` through `srcPos+length-1` in the source array are copied into positions `destPos` through `destPos+length-1`, respectively, of the destination array.

If the `src` and `dest` arguments refer to the same array object, then the copying is performed as if the components at positions `srcPos` through `srcPos+length-1` were first copied to a temporary array with `length` components and then the contents of the temporary array were copied into positions `destPos` through `destPos+length-1` of the destination array.

If `dest` is `null`, then a `NullPointerException` is thrown.

If `src` is `null`, then a `NullPointerException` is thrown and the destination array is not modified.

Otherwise, if any of the following is true, an `ArrayStoreException` is thrown and the destination is not modified:

- The `src` argument refers to an object that is not an array.
- The `dest` argument refers to an object that is not an array.

- The `src` argument and `dest` argument refer to arrays whose component types are different primitive types.
- The `src` argument refers to an array with a primitive component type and the `dest` argument refers to an array with a reference component type.
- The `src` argument refers to an array with a reference component type and the `dest` argument refers to an array with a primitive component type.

Otherwise, if any of the following is true, an `IndexOutOfBoundsException` is thrown and the destination is not modified:

- The `srcPos` argument is negative.
- The `destPos` argument is negative.
- The `length` argument is negative.
- `srcPos+length` is greater than `src.length`, the length of the source array.
- `destPos+length` is greater than `dest.length`, the length of the destination array.

Otherwise, if any actual component of the source array from position `srcPos` through `srcPos+length-1` cannot be converted to the component type of the destination array by assignment conversion, an `ArrayStoreException` is thrown. In this case, let k be the smallest nonnegative integer less than `length` such that `src[srcPos+k]` cannot be converted to the component type of the destination array; when the exception is thrown, source array components from positions `srcPos` through `srcPos+k-1` will already have been copied to destination array positions `destPos` through `destPos+k-1` and no other positions of the destination array will have been modified. (Because of the restrictions already itemized, this paragraph effectively applies only to the situation where both arrays have component types that are reference types.)

Parameters:

`src` - the source array.

`srcPos` - starting position in the source array.

`dest` - the destination array.

`destPos` - starting position in the destination data.

`length` - the number of array elements to be copied.

Throws:

[IndexOutOfBoundsException](#) - if copying would cause access of data outside array bounds.

[ArrayStoreException](#) - if an element in the `src` array could not be stored into the `dest` array because of a type mismatch.

[NullPointerException](#) - if either `src` or `dest` is `null`.

identityHashCode

```
public static int identityHashCode(Object x)
```

Returns the same hash code for the given object as would be returned by the default method `hashCode()`, whether or not the given object's class overrides `hashCode()`. The hash code for the null reference is zero.

Parameters:

x - object for which the hashCode is to be calculated

Returns:

the hashCode

Since:

JDK1.1

getProperties

```
public static Properties getProperties()
```

Determines the current system properties.

First, if there is a security manager, its `checkPropertiesAccess` method is called with no arguments. This may result in a security exception.

The current set of system properties for use by the [getProperty\(String\)](#) method is returned as a `Properties` object. If there is no current set of system properties, a set of system properties is first created and initialized. This set of system properties always includes values for the following keys:

Key	Description of Associated Value
<code>java.version</code>	Java Runtime Environment version
<code>java.vendor</code>	Java Runtime Environment vendor
<code>java.vendor.url</code>	Java vendor URL
<code>java.home</code>	Java installation directory
<code>java.vm.specification.version</code>	Java Virtual Machine specification version
<code>java.vm.specification.vendor</code>	Java Virtual Machine specification vendor
<code>java.vm.specification.name</code>	Java Virtual Machine specification name
<code>java.vm.version</code>	Java Virtual Machine implementation version
<code>java.vm.vendor</code>	Java Virtual Machine implementation vendor
<code>java.vm.name</code>	Java Virtual Machine implementation name
<code>java.specification.version</code>	Java Runtime Environment specification version

<code>java.specification.vendor</code>	Java Runtime Environment specification vendor
<code>java.specification.name</code>	Java Runtime Environment specification name
<code>java.class.version</code>	Java class format version number
<code>java.class.path</code>	Java class path
<code>java.library.path</code>	List of paths to search when loading libraries
<code>java.io.tmpdir</code>	Default temp file path
<code>java.compiler</code>	Name of JIT compiler to use
<code>java.ext.dirs</code>	Path of extension directory or directories
<code>os.name</code>	Operating system name
<code>os.arch</code>	Operating system architecture
<code>os.version</code>	Operating system version
<code>file.separator</code>	File separator ("/" on UNIX)
<code>path.separator</code>	Path separator (":" on UNIX)
<code>line.separator</code>	Line separator ("\n" on UNIX)
<code>user.name</code>	User's account name
<code>user.home</code>	User's home directory
<code>user.dir</code>	User's current working directory

Multiple paths in a system property value are separated by the path separator character of the platform.

Note that even if the security manager does not permit the `getProperties` operation, it may choose to permit the [getProperty\(String\)](#) operation.

Returns:

the system properties

Throws:

[SecurityException](#) - if a security manager exists and its `checkPropertiesAccess` method doesn't allow access to the system properties.

See Also:

[setProperty\(java.util.Properties\)](#), [SecurityException](#), [SecurityManager.checkPropertiesAccess\(\)](#), [Properties](#)

setProperty

```
public static void setProperty(Properties props)
```

Sets the system properties to the `Properties` argument.

First, if there is a security manager, its `checkPropertiesAccess` method is called with no arguments. This may result in a security exception.

The argument becomes the current set of system properties for use by the [getProperty\(`String`\)](#) method. If the argument is `null`, then the current set of system properties is forgotten.

Parameters:

`props` - the new system properties.

Throws:

[SecurityException](#) - if a security manager exists and its `checkPropertiesAccess` method doesn't allow access to the system properties.

See Also:

[getProperties\(\)](#), [Properties](#), [SecurityException](#),
[SecurityManager.checkPropertiesAccess\(\)](#)

getProperty

```
public static String getProperty(String key)
```

Gets the system property indicated by the specified key.

First, if there is a security manager, its `checkPropertyAccess` method is called with the key as its argument. This may result in a `SecurityException`.

If there is no current set of system properties, a set of system properties is first created and initialized in the same manner as for the `getProperties` method.

Parameters:

`key` - the name of the system property.

Returns:

the string value of the system property, or `null` if there is no property with that key.

Throws:

[SecurityException](#) - if a security manager exists and its `checkPropertyAccess` method doesn't allow access to the specified system property.

[NullPointerException](#) - if `key` is `null`.

[IllegalArgumentException](#) - if `key` is empty.

See Also:

[setProperty\(`java.lang.String`, `java.lang.String`\)](#),
[SecurityException](#), [SecurityManager.checkPropertyAccess\(`java.`](#)

[lang.String](#)), [getProperties\(\)](#)

getProperty

```
public static String getProperty(String key,  
                                String def)
```

Gets the system property indicated by the specified key.

First, if there is a security manager, its `checkPropertyAccess` method is called with the `key` as its argument.

If there is no current set of system properties, a set of system properties is first created and initialized in the same manner as for the `getProperties` method.

Parameters:

`key` - the name of the system property.
`def` - a default value.

Returns:

the string value of the system property, or the default value if there is no property with that key.

Throws:

[SecurityException](#) - if a security manager exists and its `checkPropertyAccess` method doesn't allow access to the specified system property.
[NullPointerException](#) - if `key` is `null`.
[IllegalArgumentException](#) - if `key` is empty.

See Also:

[setProperty\(java.lang.String, java.lang.String\)](#),
[SecurityManager.checkPropertyAccess\(java.lang.String\)](#),
[getProperties\(\)](#)

setProperty

```
public static String setProperty(String key,  
                                String value)
```

Sets the system property indicated by the specified key.

First, if a security manager exists, its `SecurityManager.checkPermission` method is called with a `PropertyPermission(key, "write")` permission. This may result in a `SecurityException` being thrown. If no exception is thrown, the specified property is set to the given value.

Parameters:

`key` - the name of the system property.

`value` - the value of the system property.

Returns:

the previous value of the system property, or `null` if it did not have one.

Throws:

[SecurityException](#) - if a security manager exists and its `checkPermission` method doesn't allow setting of the specified property.

[NullPointerException](#) - if `key` or `value` is `null`.

[IllegalArgumentException](#) - if `key` is empty.

Since:

1.2

See Also:

[getProperty\(java.lang.String\)](#), [getProperty\(java.lang.String\)](#), [getProperty\(java.lang.String, java.lang.String\)](#), [PropertyPermission](#), [SecurityManager.checkPermission\(java.security.Permission\)](#)

clearProperty

```
public static String clearProperty(String key)
```

Removes the system property indicated by the specified key.

First, if a security manager exists, its `SecurityManager.checkPermission` method is called with a `PropertyPermission(key, "write")` permission. This may result in a `SecurityException` being thrown. If no exception is thrown, the specified property is removed.

Parameters:

`key` - the name of the system property to be removed.

Returns:

the previous string value of the system property, or `null` if there was no property with that key.

Throws:

[SecurityException](#) - if a security manager exists and its `checkPropertyAccess` method doesn't allow access to the specified system property.

[NullPointerException](#) - if key is null.

[IllegalArgumentException](#) - if key is empty.

Since:

1.5

See Also:

[getProperty\(java.lang.String\)](#), [setProperty\(java.lang.String, java.lang.String\)](#), [Properties](#), [SecurityException](#), [SecurityManager.checkPropertiesAccess\(\)](#)

getenv

```
public static String getenv(String name)
```

Gets the value of the specified environment variable. An environment variable is a system-dependent external named value.

If a security manager exists, its [checkPermission](#) method is called with a [RuntimePermission](#)("getenv."+name) permission. This may result in a [SecurityException](#) being thrown. If no exception is thrown the value of the variable `name` is returned.

System properties and *environment variables* are both conceptually mappings between names and values. Both mechanisms can be used to pass user-defined information to a Java process. Environment variables have a more global effect, because they are visible to all descendants of the process which defines them, not just the immediate Java subprocess. They can have subtly different semantics, such as case insensitivity, on different operating systems. For these reasons, environment variables are more likely to have unintended side effects. It is best to use system properties where possible. Environment variables should be used when a global effect is desired, or when an external system interface requires an environment variable (such as PATH).

On UNIX systems the alphabetic case of `name` is typically significant, while on Microsoft Windows systems it is typically not. For example, the expression `System.getenv("FOO").equals(System.getenv("foo"))` is likely to be true on Microsoft Windows.

Parameters:

`name` - the name of the environment variable

Returns:

the string value of the variable, or null if the variable is not defined in the system environment

Throws:

[NullPointerException](#) - if name is null

[SecurityException](#) - if a security manager exists and its [checkPermission](#) method doesn't allow access to the environment variable name

See Also:

[getenv\(\)](#), [ProcessBuilder.environment\(\)](#)

getenv

```
public static Map<String, String> getenv()
```

Returns an unmodifiable string map view of the current system environment. The environment is a system-dependent mapping from names to values which is passed from parent to child processes.

If the system does not support environment variables, an empty map is returned.

The returned map will never contain null keys or values. Attempting to query the presence of a null key or value will throw a [NullPointerException](#). Attempting to query the presence of a key or value which is not of type [String](#) will throw a [ClassCastException](#).

The returned map and its collection views may not obey the general contract of the [Object.equals\(java.lang.Object\)](#) and [Object.hashCode\(\)](#) methods.

The returned map is typically case-sensitive on all platforms.

If a security manager exists, its [checkPermission](#) method is called with a [RuntimePermission\("getenv.*"\)](#) permission. This may result in a [SecurityException](#) being thrown.

When passing information to a Java subprocess, [system properties](#) are generally preferred over environment variables.

Returns:

the environment as a map of variable names to values

Throws:

[SecurityException](#) - if a security manager exists and its [checkPermission](#) method doesn't allow access to the process environment

Since:

1.5

See Also:

[getenv\(String\)](#), [ProcessBuilder.environment\(\)](#)

exit

```
public static void exit(int status)
```

Terminates the currently running Java Virtual Machine. The argument serves as a status code; by convention, a nonzero status code indicates abnormal termination.

This method calls the `exit` method in class `Runtime`. This method never returns normally.

The call `System.exit(n)` is effectively equivalent to the call:

```
Runtime.getRuntime().exit(n)
```

Parameters:

`status` - exit status.

Throws:

[SecurityException](#) - if a security manager exists and its `checkExit` method doesn't allow exit with the specified status.

See Also:

[Runtime.exit\(int\)](#)

gc

```
public static void gc()
```

Runs the garbage collector.

Calling the `gc` method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all discarded objects.

The call `System.gc()` is effectively equivalent to the call:

```
Runtime.getRuntime().gc()
```

See Also:[Runtime.gc\(\)](#)

runFinalization

```
public static void runFinalization()
```

Runs the finalization methods of any objects pending finalization.

Calling this method suggests that the Java Virtual Machine expend effort toward running the `finalize` methods of objects that have been found to be discarded but whose `finalize` methods have not yet been run. When control returns from the method call, the Java Virtual Machine has made a best effort to complete all outstanding finalizations.

The call `System.runFinalization()` is effectively equivalent to the call:

```
Runtime.getRuntime().runFinalization()
```

See Also:[Runtime.runFinalization\(\)](#)

runFinalizersOnExit

```
public static void runFinalizersOnExit(boolean value)
```

Deprecated. *This method is inherently unsafe. It may result in finalizers being called on live objects while other threads are concurrently manipulating those objects, resulting in erratic behavior or deadlock.*

Enable or disable finalization on exit; doing so specifies that the finalizers of all objects that have finalizers that have not yet been automatically invoked are to be run before the Java runtime exits. By default, finalization on exit is disabled.

If there is a security manager, its `checkExit` method is first called with 0 as its argument to ensure the exit is allowed. This could result in a `SecurityException`.

Parameters:

`value` - indicating enabling or disabling of finalization

Throws:

[SecurityException](#) - if a security manager exists and its `checkExit` method doesn't allow the exit.

Since:

JDK1.1

See Also:

[Runtime.exit\(int\)](#), [Runtime.gc\(\)](#), [SecurityManager.checkExit\(int\)](#)

load

```
public static void load(String filename)
```

Loads a code file with the specified filename from the local file system as a dynamic library. The filename argument must be a complete path name.

The call `System.load(name)` is effectively equivalent to the call:

```
Runtime.getRuntime().load(name)
```

Parameters:

filename - the file to load.

Throws:

[SecurityException](#) - if a security manager exists and its `checkLink` method doesn't allow loading of the specified dynamic library

[UnsatisfiedLinkError](#) - if the file does not exist.

[NullPointerException](#) - if filename is null

See Also:

[Runtime.load\(java.lang.String\)](#), [SecurityManager.checkLink\(java.lang.String\)](#)

loadLibrary

```
public static void loadLibrary(String libname)
```

Loads the system library specified by the `libname` argument. The manner in which a library name is mapped to the actual system library is system dependent.

The call `System.loadLibrary(name)` is effectively equivalent to the call

```
Runtime.getRuntime().loadLibrary(name)
```

Parameters:

`libname` - the name of the library.

Throws:

[SecurityException](#) - if a security manager exists and its `checkLink` method doesn't allow loading of the specified dynamic library

[UnsatisfiedLinkError](#) - if the library does not exist.

[NullPointerException](#) - if `libname` is `null`

See Also:

[Runtime.loadLibrary\(java.lang.String\)](#), [SecurityManager.checkLink\(java.lang.String\)](#)

mapLibraryName

```
public static String mapLibraryName(String libname)
```

Maps a library name into a platform-specific string representing a native library.

Parameters:

`libname` - the name of the library.

Returns:

a platform-dependent native library name.

Throws:

[NullPointerException](#) - if `libname` is `null`

Since:

1.2

See Also:

[loadLibrary\(java.lang.String\)](#), [ClassLoader.findLibrary\(java.lang.String\)](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.5.0*

[PREV CLASS](#) [NEXT CLASS](#)

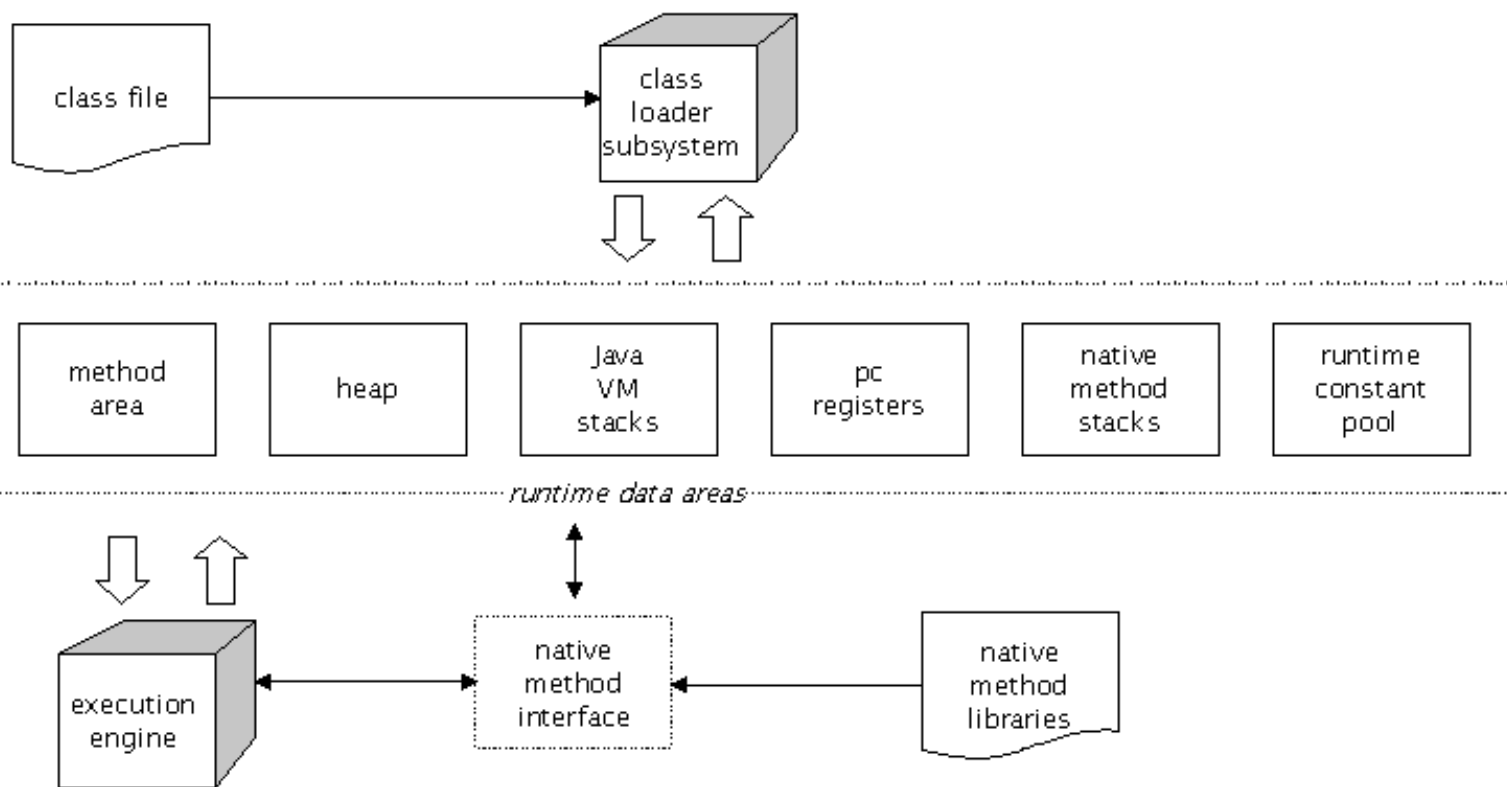
[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | [FIELD](#) | CONSTR | [METHOD](#) DETAIL: [FIELD](#) | CONSTR | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).



CHAPTER 3

The Structure of the Java Virtual Machine

This book specifies an abstract machine. It does not document any particular implementation of the Java virtual machine, including Sun Microsystems'.

To implement the Java virtual machine correctly, you need only be able to read the `class` file format and correctly perform the operations specified therein. Implementation details that are not part of the Java virtual machine's specification would unnecessarily constrain the creativity of implementors. For example, the memory layout of run-time data areas, the garbage-collection algorithm used, and any internal optimization of the Java virtual machine instructions (for example, translating them into machine code) are left to the discretion of the implementor.

3.1 The `class` File Format

Compiled code to be executed by the Java virtual machine is represented using a hardware- and operating system-independent binary format, typically (but not necessarily) stored in a file, known as the `class` file format. The `class` file format precisely defines the representation of a class or interface, including details such as byte ordering that might be taken for granted in a platform-specific object file format.

[Chapter 4, "The `class` File Format"](#), covers the `class` file format in detail.

3.2 Data Types

Like the Java programming language, the Java virtual machine operates on two kinds of types: *primitive types* and *reference types*. There are, correspondingly, two kinds of values that can be stored in variables, passed as arguments, returned by methods, and operated upon: *primitive values* and *reference values*.

The Java virtual machine expects that nearly all type checking is done prior to run time, typically by a

compiler, and does not have to be done by the Java virtual machine itself. Values of primitive types need not be tagged or otherwise be inspectable to determine their types at run time, or to be distinguished from values of reference types. Instead, the instruction set of the Java virtual machine distinguishes its operand types using instructions intended to operate on values of specific types. For instance, *iadd*, *ladd*, *fadd*, and *dadd* are all Java virtual machine instructions that add two numeric values and produce numeric results, but each is specialized for its operand type: `int`, `long`, `float`, and `double`, respectively. For a summary of type support in the Java virtual machine instruction set, see [§3.11.1](#).

The Java virtual machine contains explicit support for objects. An object is either a dynamically allocated class instance or an array. A reference to an object is considered to have Java virtual machine type `reference`. Values of type `reference` can be thought of as pointers to objects. More than one reference to an object may exist. Objects are always operated on, passed, and tested via values of type `reference`.

3.3 Primitive Types and Values

The primitive data types supported by the Java virtual machine are the *numeric types*, the `boolean` type ([§3.3.4](#)),¹ and the `returnAddress` type ([§3.3.3](#)). The numeric types consist of the *integral types* ([§3.3.1](#)) and the *floating-point types* ([§3.3.2](#)). The integral types are:

- `byte`, whose values are 8-bit signed two's-complement integers
- `short`, whose values are 16-bit signed two's-complement integers
- `int`, whose values are 32-bit signed two's-complement integers
- `long`, whose values are 64-bit signed two's-complement integers
- `char`, whose values are 16-bit unsigned integers representing Unicode characters ([§2.1](#))

The floating-point types are:

- `float`, whose values are elements of the float value set or, where supported, the float-extended-exponent value set
- `double`, whose values are elements of the double value set or, where supported, the double-extended-exponent value set

The values of the `boolean` type encode the truth values `true` and `false`.

The values of the `returnAddress` type are pointers to the opcodes of Java virtual machine instructions. Of the primitive types only the `returnAddress` type is not directly associated with a Java programming language type.

3.3.1 Integral Types and Values

The values of the integral types of the Java virtual machine are the same as those for the integral types of the Java programming language ([§2.4.1](#)):

- For `byte`, from -128 to 127 (-2^7 to 2^7-1), inclusive
- For `short`, from -32768 to 32767 (-2^{15} to $2^{15}-1$), inclusive
- For `int`, from -2147483648 to 2147483647 (-2^{31} to $2^{31}-1$), inclusive
- For `long`, from -9223372036854775808 to 9223372036854775807 (-2^{63} to $2^{63}-1$), inclusive
- For `char`, from 0 to 65535 inclusive

3.3.2 Floating-Point Types, Value Sets, and Values

The floating-point types are `float` and `double`, which are conceptually associated with the 32-bit single-precision and 64-bit double-precision format IEEE 754 values and operations as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std. 754-1985 (IEEE, New York).

The IEEE 754 standard includes not only positive and negative sign-magnitude numbers, but also positive and negative zeros, positive and negative *infinities*, and a special *Not-a-Number value* (hereafter abbreviated as "NaN"). The NaN value is used to represent the result of certain invalid operations such as dividing zero by zero.

Every implementation of the Java virtual machine is required to support two standard sets of floating-point values, called the *float value set* and the *double value set*. In addition, an implementation of the Java virtual machine may, at its option, support either or both of two extended-exponent floating-point value sets, called the *float-extended-exponent value set* and the *double-extended-exponent value set*. These extended-exponent value sets may, under certain circumstances, be used instead of the standard value sets to represent the values of type `float` or `double`.

The finite nonzero values of any floating-point value set can all be expressed in the form $s \cdot m \cdot 2^{e-N}$ ($+1$), where s is $+1$ or -1 , m is a positive integer less than 2^N , and e is an integer between $E_{\min} = -(2^K - 1 - 2)$ and $E_{\max} = 2^K - 1 - 1$, inclusive, and where N and K are parameters that depend on the value set. Some values can be represented in this form in more than one way; for example, supposing that a value v in a value set might be represented in this form using certain values for s , m , and e , then if it happened that m were even and e were less than $2^K - 1$, one could halve m and increase e by 1 to

produce a second representation for the same value v . A representation in this form is called *normalized* if $m \geq 2^{N-1}$; otherwise the representation is said to be *denormalized*. If a value in a value set cannot be represented in such a way that $m \geq 2^{N-1}$, then the value is said to be a *denormalized value*, because it has no normalized representation.

The constraints on the parameters N and K (and on the derived parameters $E_{\min}$ and $E_{\max}$) for the two required and two optional floating-point value sets are summarized in [Table 3.1](#).

<i>Parameter</i>	<i>float</i>	<i>float-extended-exponent</i>	<i>double</i>	<i>double-extended-exponent</i>
N	24	24	53	53
K	8	≥ 11	11	≥ 15
$E_{\max}$	+127	$\geq +1023$	+1023	$\geq +16383$
$E_{\min}$	-126	≤ -1022	-1022	≤ -16382

Where one or both extended-exponent value sets are supported by an implementation, then for each supported extended-exponent value set there is a specific implementation-dependent constant K , whose value is constrained by [Table 3.1](#); this value K in turn dictates the values for $E_{\min}$ and $E_{\max}$.

Each of the four value sets includes not only the finite nonzero values that are ascribed to it above, but also the five values positive zero, negative zero, positive infinity, negative infinity, and NaN.

Note that the constraints in [Table 3.1](#) are designed so that every element of the float value set is necessarily also an element of the float-extended-exponent value set, the double value set, and the double-extended-exponent value set. Likewise, each element of the double value set is necessarily also an element of the double-extended-exponent value set. Each extended-exponent value set has a larger range of exponent values than the corresponding standard value set, but does not have more precision.

The elements of the float value set are exactly the values that can be represented using the single floating-point format defined in the IEEE 754 standard, except that there is only one NaN value (IEEE 754 specifies $2^{24} - 2$ distinct NaN values). The elements of the double value set are exactly the values that can be represented using the double floating-point format defined in the IEEE 754 standard, except that there is only one NaN value (IEEE 754 specifies $2^{53} - 2$ distinct NaN values). Note, however, that the elements of the float-extended-exponent and double-extended-exponent value sets defined here do *not* correspond to the values that be represented using IEEE 754 single extended and double extended formats, respectively. This specification does not mandate a specific representation for the values of the floating-point value sets except where floating-point values must be represented in the `class` file format ([§4.4.4](#), [§4.4.5](#)).

The float, float-extended-exponent, double, and double-extended-exponent value sets are not types. It

is always correct for an implementation of the Java virtual machine to use an element of the float value set to represent a value of type `float`; however, it may be permissible in certain contexts for an implementation to use an element of the float-extended-exponent value set instead. Similarly, it is always correct for an implementation to use an element of the double value set to represent a value of type `double`; however, it may be permissible in certain contexts for an implementation to use an element of the double-extended-exponent value set instead.

Except for NaNs, values of the floating-point value sets are *ordered*. When arranged from smallest to largest, they are negative infinity, negative finite values, positive and negative zero, positive finite values, and positive infinity.

Floating-point positive zero and floating-point negative zero compare as equal, but there are other operations that can distinguish them; for example, dividing `1.0` by `0.0` produces positive infinity, but dividing `1.0` by `-0.0` produces negative infinity.

NaNs are *unordered*, so numerical comparisons and tests for numerical equality have the value `false` if either or both of their operands are NaN. In particular, a test for numerical equality of a value against itself has the value `false` if and only if the value is NaN. A test for numerical inequality has the value `true` if either operand is NaN.

3.3.3 The `returnAddress` Type and Values

The `returnAddress` type is used by the Java virtual machine's `jsr`, `ret`, and `jsr_w` instructions. The values of the `returnAddress` type are pointers to the opcodes of Java virtual machine instructions. Unlike the numeric primitive types, the `returnAddress` type does not correspond to any Java programming language type and cannot be modified by the running program.

3.3.4 The `boolean` Type

Although the Java virtual machine defines a `boolean` type, it only provides very limited support for it. There are no Java virtual machine instructions solely dedicated to operations on `boolean` values. Instead, expressions in the Java programming language that operate on `boolean` values are compiled to use values of the Java virtual machine `int` data type.

The Java virtual machine does directly support `boolean` arrays. Its `newarray` instruction enables creation of `boolean` arrays. Arrays of type `boolean` are accessed and modified using the `byte` array instructions `baload` and `bastore`.²

The Java virtual machine encodes `boolean` array components using `1` to represent `true` and `0` to represent `false`. Where Java programming language `boolean` values are mapped by compilers to values of Java virtual machine type `int`, the compilers must use the same encoding.

3.4 Reference Types and Values

There are three kinds of `reference` types: class types, array types, and interface types. Their values are references to dynamically created class instances, arrays, or class instances or arrays that implement interfaces, respectively. A `reference` value may also be the special null reference, a reference to no object, which will be denoted here by `null`. The `null` reference initially has no runtime type, but may be cast to any type ([§2.4](#)).

The Java virtual machine specification does not mandate a concrete value encoding `null`.

3.5 Runtime Data Areas

The Java virtual machine defines various runtime data areas that are used during execution of a program. Some of these data areas are created on Java virtual machine start-up and are destroyed only when the Java virtual machine exits. Other data areas are per thread. Per-thread data areas are created when a thread is created and destroyed when the thread exits.

3.5.1 The pc Register

The Java virtual machine can support many threads of execution at once ([§2.19](#)). Each Java virtual machine thread has its own `pc` (program counter) register. At any point, each Java virtual machine thread is executing the code of a single method, the current method ([§3.6](#)) for that thread. If that method is not `native`, the `pc` register contains the address of the Java virtual machine instruction currently being executed. If the method currently being executed by the thread is `native`, the value of the Java virtual machine's `pc` register is undefined. The Java virtual machine's `pc` register is wide enough to hold a `returnAddress` or a native pointer on the specific platform.

3.5.2 Java Virtual Machine Stacks

Each Java virtual machine thread has a private *Java virtual machine stack*, created at the same time as the thread.³ A Java virtual machine stack stores frames ([§3.6](#)). A Java virtual machine stack is analogous to the stack of a conventional language such as C: it holds local variables and partial results, and plays a part in method invocation and return. Because the Java virtual machine stack is never manipulated directly except to push and pop frames, frames may be heap allocated. The memory for a Java virtual machine stack does not need to be contiguous.

The Java virtual machine specification permits Java virtual machine stacks either to be of a fixed size or to dynamically expand and contract as required by the computation. If the Java virtual machine stacks are of a fixed size, the size of each Java virtual machine stack may be chosen independently when that stack is created. A Java virtual machine implementation may provide the programmer or the user control over the initial size of Java virtual machine stacks, as well as, in the case of

dynamically expanding or contracting Java virtual machine stacks, control over the maximum and minimum sizes.^{[4](#)}

The following exceptional conditions are associated with Java virtual machine stacks:

- If the computation in a thread requires a larger Java virtual machine stack than is permitted, the Java virtual machine throws a `StackOverflowError`.
- If Java virtual machine stacks can be dynamically expanded, and expansion is attempted but insufficient memory can be made available to effect the expansion, or if insufficient memory can be made available to create the initial Java virtual machine stack for a new thread, the Java virtual machine throws an `OutOfMemoryError`.

3.5.3 Heap

The Java virtual machine has a *heap* that is shared among all Java virtual machine threads. The heap is the runtime data area from which memory for all class instances and arrays is allocated.

The heap is created on virtual machine start-up. Heap storage for objects is reclaimed by an automatic storage management system (known as a *garbage collector*); objects are never explicitly deallocated. The Java virtual machine assumes no particular type of automatic storage management system, and the storage management technique may be chosen according to the implementor's system requirements. The heap may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger heap becomes unnecessary. The memory for the heap does not need to be contiguous.

A Java virtual machine implementation may provide the programmer or the user control over the initial size of the heap, as well as, if the heap can be dynamically expanded or contracted, control over the maximum and minimum heap size.^{[5](#)}

The following exceptional condition is associated with the heap:

- If a computation requires more heap than can be made available by the automatic storage management system, the Java virtual machine throws an `OutOfMemoryError`.

3.5.4 Method Area

The Java virtual machine has a *method area* that is shared among all Java virtual machine threads. The method area is analogous to the storage area for compiled code of a conventional language or analogous to the "text" segment in a UNIX process. It stores per-class structures such as the runtime constant pool, field and method data, and the code for methods and constructors, including the special methods ([§3.9](#)) used in class and instance initialization and interface type initialization.

The method area is created on virtual machine start-up. Although the method area is logically part of

the heap, simple implementations may choose not to either garbage collect or compact it. This version of the Java virtual machine specification does not mandate the location of the method area or the policies used to manage compiled code. The method area may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger method area becomes unnecessary. The memory for the method area does not need to be contiguous.

A Java virtual machine implementation may provide the programmer or the user control over the initial size of the method area, as well as, in the case of a varying-size method area, control over the maximum and minimum method area size.⁶

The following exceptional condition is associated with the method area:

- If memory in the method area cannot be made available to satisfy an allocation request, the Java virtual machine throws an `OutOfMemoryError`.

3.5.5 Runtime Constant Pool

A *runtime constant pool* is a per-class or per-interface runtime representation of the `constant_pool` table in a `class` file (§4.4). It contains several kinds of constants, ranging from numeric literals known at compile time to method and field references that must be resolved at run time. The runtime constant pool serves a function similar to that of a symbol table for a conventional programming language, although it contains a wider range of data than a typical symbol table.

Each runtime constant pool is allocated from the Java virtual machine's method area (§3.5.4). The runtime constant pool for a class or interface is constructed when the class or interface is created (§5.3) by the Java virtual machine.

The following exceptional condition is associated with the construction of the runtime constant pool for a class or interface:

- When creating a class or interface, if the construction of the runtime constant pool requires more memory than can be made available in the method area of the Java virtual machine, the Java virtual machine throws an `OutOfMemoryError`.

See [Chapter 5](#) for information about the construction of the runtime constant pool.

3.5.6 Native Method Stacks

An implementation of the Java virtual machine may use conventional stacks, colloquially called "C stacks," to support `native` methods, methods written in a language other than the Java programming language. Native method stacks may also be used by the implementation of an interpreter for the Java virtual machine's instruction set in a language such as C. Java virtual machine implementations that cannot load `native` methods and that do not themselves rely on conventional stacks need not supply native method stacks. If supplied, native method stacks are typically allocated

per thread when each thread is created.

The Java virtual machine specification permits native method stacks either to be of a fixed size or to dynamically expand and contract as required by the computation. If the native method stacks are of a fixed size, the size of each native method stack may be chosen independently when that stack is created. In any case, a Java virtual machine implementation may provide the programmer or the user control over the initial size of the native method stacks. In the case of varying-size native method stacks, it may also make available control over the maximum and minimum method stack sizes.⁷

The following exceptional conditions are associated with native method stacks:

- If the computation in a thread requires a larger native method stack than is permitted, the Java virtual machine throws a `StackOverflowError`.
- If native method stacks can be dynamically expanded and native method stack expansion is attempted but insufficient memory can be made available, or if insufficient memory can be made available to create the initial native method stack for a new thread, the Java virtual machine throws an `OutOfMemoryError`.

3.6 Frames

A *frame* is used to store data and partial results, as well as to perform dynamic linking , return values for methods, and dispatch exceptions.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes, whether that completion is normal or abrupt (it throws an uncaught exception). Frames are allocated from the Java virtual machine stack ([§3.5.2](#)) of the thread creating the frame. Each frame has its own array of local variables ([§3.6.1](#)), its own operand stack ([§3.6.2](#)), and a reference to the runtime constant pool ([§3.5.5](#)) of the class of the current method.

The sizes of the local variable array and the operand stack are determined at compile time and are supplied along with the code for the method associated with the frame ([§4.7.3](#)). Thus the size of the frame data structure depends only on the implementation of the Java virtual machine, and the memory for these structures can be allocated simultaneously on method invocation.

Only one frame, the frame for the executing method, is active at any point in a given thread of control. This frame is referred to as the *current frame*, and its method is known as the *current method*. The class in which the current method is defined is the *current class*. Operations on local variables and the operand stack are typically with reference to the current frame.

A frame ceases to be current if its method invokes another method or if its method completes. When a method is invoked, a new frame is created and becomes current when control transfers to the new

method. On method return, the current frame passes back the result of its method invocation, if any, to the previous frame. The current frame is then discarded as the previous frame becomes the current one.

Note that a frame created by a thread is local to that thread and cannot be referenced by any other thread.

3.6.1 Local Variables

Each frame ([§3.6](#)) contains an array of variables known as its *local variables*. The length of the local variable array of a frame is determined at compile time and supplied in the binary representation of a class or interface along with the code for the method associated with the frame ([§4.7.3](#)).

A single local variable can hold a value of type `boolean`, `byte`, `char`, `short`, `int`, `float`, `reference`, or `returnAddress`. A pair of local variables can hold a value of type `long` or `double`.

Local variables are addressed by indexing. The index of the first local variable is zero. An integer is be considered to be an index into the local variable array if and only if that integer is between zero and one less than the size of the local variable array.

A value of type `long` or type `double` occupies two consecutive local variables. Such a value may only be addressed using the lesser index. For example, a value of type `double` stored in the local variable array at index n actually occupies the local variables with indices n and $n + 1$; however, the local variable at index $n + 1$ cannot be loaded from. It can be stored into. However, doing so invalidates the contents of local variable n .

The Java virtual machine does not require n to be even. In intuitive terms, values of types `double` and `long` need not be 64-bit aligned in the local variables array. Implementors are free to decide the appropriate way to represent such values using the two local variables reserved for the value.

The Java virtual machine uses local variables to pass parameters on method invocation. On class method invocation any parameters are passed in consecutive local variables starting from local variable 0 . On instance method invocation, local variable 0 is always used to pass a reference to the object on which the instance method is being invoked (`this` in the Java programming language). Any parameters are subsequently passed in consecutive local variables starting from local variable 1 .

3.6.2 Operand Stacks

Each frame ([§3.6](#)) contains a last-in-first-out (LIFO) stack known as its *operand stack*. The maximum depth of the operand stack of a frame is determined at compile time and is supplied along with the code for the method associated with the frame ([§4.7.3](#)).

Where it is clear by context, we will sometimes refer to the operand stack of the current frame as

simply the operand stack.

The operand stack is empty when the frame that contains it is created. The Java virtual machine supplies instructions to load constants or values from local variables or fields onto the operand stack. Other Java virtual machine instructions take operands from the operand stack, operate on them, and push the result back onto the operand stack. The operand stack is also used to prepare parameters to be passed to methods and to receive method results.

For example, the *iadd* instruction adds two `int` values together. It requires that the `int` values to be added be the top two values of the operand stack, pushed there by previous instructions. Both of the `int` values are popped from the operand stack. They are added, and their sum is pushed back onto the operand stack. Subcomputations may be nested on the operand stack, resulting in values that can be used by the encompassing computation.

Each entry on the operand stack can hold a value of any Java virtual machine type, including a value of type `long` or type `double`.

Values from the operand stack must be operated upon in ways appropriate to their types. It is not possible, for example, to push two `int` values and subsequently treat them as a `long` or to push two `float` values and subsequently add them with an *iadd* instruction. A small number of Java virtual machine instructions (the *dup* instructions and *swap*) operate on runtime data areas as raw values without regard to their specific types; these instructions are defined in such a way that they cannot be used to modify or break up individual values. These restrictions on operand stack manipulation are enforced through `class` file verification ([§4.9](#)).

At any point in time an operand stack has an associated depth, where a value of type `long` or `double` contributes two units to the depth and a value of any other type contributes one unit.

3.6.3 Dynamic Linking

Each frame ([§3.6](#)) contains a reference to the runtime constant pool ([§3.5.5](#)) for the type of the current method to support *dynamic linking* of the method code. The `class` file code for a method refers to methods to be invoked and variables to be accessed via symbolic references. Dynamic linking translates these symbolic method references into concrete method references, loading classes as necessary to resolve as-yet-undefined symbols, and translates variable accesses into appropriate offsets in storage structures associated with the runtime location of these variables.

This late binding of the methods and variables makes changes in other classes that a method uses less likely to break this code.

3.6.4 Normal Method Invocation Completion

A method invocation *completes normally* if that invocation does not cause an exception ([§2.16](#), [§3.10](#)) to be thrown, either directly from the Java virtual machine or as a result of executing an

explicit `throw` statement. If the invocation of the current method completes normally, then a value may be returned to the invoking method. This occurs when the invoked method executes one of the return instructions ([§3.11.8](#)), the choice of which must be appropriate for the type of the value being returned (if any).

The current frame ([§3.6](#)) is used in this case to restore the state of the invoker, including its local variables and operand stack, with the program counter of the invoker appropriately incremented to skip past the method invocation instruction. Execution then continues normally in the invoking method's frame with the returned value (if any) pushed onto the operand stack of that frame.

3.6.5 Abrupt Method Invocation Completion

A method invocation *completes abruptly* if execution of a Java virtual machine instruction within the method causes the Java virtual machine to throw an exception ([§2.16](#), [§3.10](#)), and that exception is not handled within the method. Execution of an *athrow* instruction also causes an exception to be explicitly thrown and, if the exception is not caught by the current method, results in abrupt method invocation completion. A method invocation that completes abruptly never returns a value to its invoker.

3.6.6 Additional Information

A frame may be extended with additional implementation-specific information, such as debugging information.

3.7 Representation of Objects

The Java virtual machine does not mandate any particular internal structure for objects.[8](#)

3.8 Floating-Point Arithmetic

The Java virtual machine incorporates a subset of the floating-point arithmetic specified in *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std. 754-1985, New York).

3.8.1 Java Virtual Machine Floating-Point Arithmetic and IEEE 754

The key differences between the floating-point arithmetic supported by the Java virtual machine and the IEEE 754 standard are:

- The floating-point operations of the Java virtual machine do not throw exceptions, trap, or

otherwise signal the IEEE 754 exceptional conditions of invalid operation, division by zero, overflow, underflow, or inexact. The Java virtual machine has no signaling NaN value.

- The Java virtual machine does not support IEEE 754 signaling floating-point comparisons.
- The rounding operations of the Java virtual machine always use IEEE 754 round to nearest mode. Inexact results are rounded to the nearest representable value, with ties going to the value with a zero least-significant bit. This is the IEEE 754 default mode. But Java virtual machine instructions that convert values of floating-point types to values of integral types round toward zero. The Java virtual machine does not give any means to change the floating-point rounding mode.
- The Java virtual machine does not support either the IEEE 754 single extended or double extended format, except insofar as the double and double-extended-exponent value sets may be said to support the single extended format. The float-extended-exponent and double-extended-exponent value sets, which may optionally be supported, do not correspond to the values of the IEEE 754 extended formats: the IEEE 754 extended formats require extended precision as well as extended exponent range.

3.8.2 Floating-Point Modes

Every method has a *floating-point mode*, which is either *FP-strict* or *not FP-strict*. The floating-point mode of a method is determined by the setting of the `ACC_STRICT` bit of the `access_flags` item of the `method_info` structure ([§4.6](#)) defining the method. A method for which this bit is set is FP-strict; otherwise, the method is not FP-strict.

Note that this mapping of the `ACC_STRICT` bit implies that methods in classes compiled by a compiler that predates the Java 2 platform, v1.2, are effectively not FP-strict.

We will refer to an operand stack as having a given floating-point mode when the method whose invocation created the frame containing the operand stack has that floating-point mode. Similarly, we will refer to a Java virtual machine instruction as having a given floating-point mode when the method containing that instruction has that floating-point mode.

If a float-extended-exponent value set is supported ([§3.3.2](#)), values of type `float` on an operand stack that is not FP-strict may range over that value set except where prohibited by value set conversion ([§3.8.3](#)). If a double-extended-exponent value set is supported ([§3.3.2](#)), values of type `double` on an operand stack that is not FP-strict may range over that value set except where prohibited by value set conversion.

In all other contexts, whether on the operand stack or elsewhere, and regardless of floating-point mode, floating-point values of type `float` and `double` may only range over the float value set and double value set, respectively. In particular, class and instance fields, array elements, local variables, and method parameters may only contain values drawn from the standard value sets.

3.8.3 Value Set Conversion

An implementation of the Java virtual machine that supports an extended floating-point value set is permitted or required, under specified circumstances, to map a value of the associated floating-point type between the extended and the standard value sets. Such a *value set conversion* is not a type conversion, but a mapping between the value sets associated with the same type.

Where value set conversion is indicated, an implementation is permitted to perform one of the following operations on a value:

- If the value is of type `float` and is not an element of the float value set, it maps the value to the nearest element of the float value set.
- If the value is of type `double` and is not an element of the double value set, it maps the value to the nearest element of the double value set.

In addition, where value set conversion is indicated certain operations are required:

- Suppose execution of a Java virtual machine instruction that is not FP-strict causes a value of type `float` to be pushed onto an operand stack that is FP-strict, passed as a parameter, or stored into a local variable, a field, or an element of an array. If the value is not an element of the float value set, it maps the value to the nearest element of the float value set.
- Suppose execution of a Java virtual machine instruction that is not FP-strict causes a value of type `double` to be pushed onto an operand stack that is FP-strict, passed as a parameter, or stored into a local variable, a field, or an element of an array. If the value is not an element of the double value set, it maps the value to the nearest element of the double value set.

Such required value set conversions may occur as a result of passing a parameter of a floating-point type during method invocation, including `native` method invocation; returning a value of a floating-point type from a method that is not FP-strict to a method that is FP-strict; or storing a value of a floating-point type into a local variable, a field, or an array in a method that is not FP-strict.

Not all values from an extended-exponent value set can be mapped exactly to a value in the corresponding standard value set. If a value being mapped is too large to be represented exactly (its exponent is greater than that permitted by the standard value set), it is converted to a (positive or negative) infinity of the corresponding type. If a value being mapped is too small to be represented exactly (its exponent is smaller than that permitted by the standard value set), it is rounded to the nearest of a representable denormalized value or zero of the same sign.

Value set conversion preserves infinities and NaNs and cannot change the sign of the value being converted. Value set conversion has no effect on a value that is not of a floating-point type.

3.9 Specially Named Initialization Methods

At the level of the Java virtual machine, every constructor ([§2.12](#)) appears as an *instance initialization method* that has the special name `<init>`. This name is supplied by a compiler. Because the name `<init>` is not a valid identifier, it cannot be used directly in a program written in the Java programming language. Instance initialization methods may be invoked only within the Java virtual machine by the *invokespecial* instruction, and they may be invoked only on uninitialized class instances. An instance initialization method takes on the access permissions ([§2.7.4](#)) of the constructor from which it was derived.

A class or interface has at most one *class or interface initialization method* and is initialized ([§2.17.4](#)) by invoking that method. The initialization method of a class or interface is static and takes no arguments. It has the special name `<clinit>`. This name is supplied by a compiler. Because the name `<clinit>` is not a valid identifier, it cannot be used directly in a program written in the Java programming language. Class and interface initialization methods are invoked implicitly by the Java virtual machine; they are never invoked directly from any Java virtual machine instruction, but are invoked only indirectly as part of the class initialization process.

3.10 Exceptions

In the Java programming language, throwing an exception results in an immediate nonlocal transfer of control from the point where the exception was thrown. This transfer of control may abruptly complete, one by one, multiple statements, constructor invocations, static and field initializer evaluations, and method invocations. The process continues until a `catch` clause ([§2.16.2](#)) is found that handles the thrown value. If no such clause can be found, the current thread exits.

In cases where a `finally` clause ([§2.16.2](#)) is used, the `finally` clause is executed during the propagation of an exception thrown from the associated `try` block and any associated `catch` block, even if no `catch` clause that handles the thrown exception may be found.

As implemented by the Java virtual machine, each `catch` or `finally` clause of a method is represented by an exception handler. An exception handler specifies the range of offsets into the Java virtual machine code implementing the method for which the exception handler is active, describes the type of exception that the exception handler is able to handle, and specifies the location of the code that is to handle that exception. An exception matches an exception handler if the offset of the instruction that caused the exception is in the range of offsets of the exception handler and the exception type is the same class as or a subclass of the class of exception that the exception handler handles. When an exception is thrown, the Java virtual machine searches for a matching exception handler in the current method. If a matching exception handler is found, the system branches to the exception handling code specified by the matched handler.

If no such exception handler is found in the current method, the current method invocation completes abruptly (§3.6.5). On abrupt completion, the operand stack and local variables of the current method invocation are discarded, and its frame is popped, reinstating the frame of the invoking method. The exception is then rethrown in the context of the invoker's frame and so on, continuing up the method invocation chain. If no suitable exception handler is found before the top of the method invocation chain is reached, the execution of the thread in which the exception was thrown is terminated.

The order in which the exception handlers of a method are searched for a match is important. Within a `class` file the exception handlers for each method are stored in a table (§4.7.3). At run time, when an exception is thrown, the Java virtual machine searches the exception handlers of the current method in the order that they appear in the corresponding exception handler table in the `class` file, starting from the beginning of that table. Because `try` statements are structured, a compiler for the Java programming language can always order the entries of the exception handler table such that, for any thrown exception and any program counter value in that method, the first exception handler that matches the thrown exception corresponds to the innermost matching `catch` or `finally` clause.

Note that the Java virtual machine does not enforce nesting of or any ordering of the exception table entries of a method (§4.9.5). The exception handling semantics of the Java programming language are implemented only through cooperation with the compiler. When `class` files are generated by some other means, the defined search procedure ensures that all Java virtual machines will behave consistently.

More information on the implementation of `catch` and `finally` clauses is given in [Chapter 7](#), "[Compiling for the Java Virtual Machine](#)."

3.11 Instruction Set Summary

A Java virtual machine instruction consists of a one-byte *opcode* specifying the operation to be performed, followed by zero or more *operands* supplying arguments or data that are used by the operation. Many instructions have no operands and consist only of an opcode.

Ignoring exceptions, the inner loop of a Java virtual machine interpreter is effectively

```
do {
    fetch an opcode;
    if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
```

The number and size of the operands are determined by the opcode. If an operand is more than one byte in size, then it is stored in *big-endian* order-high-order byte first. For example, an unsigned 16-bit index into the local variables is stored as two unsigned bytes, *byte1* and *byte2*, such that its value

is

$$(byte1 \ll 8) \mid byte2$$

The bytecode instruction stream is only single-byte aligned. The two exceptions are the *tableswitch* and *lookupswitch* instructions, which are padded to force internal alignment of some of their operands on 4-byte boundaries.

The decision to limit the Java virtual machine opcode to a byte and to forgo data alignment within compiled code reflects a conscious bias in favor of compactness, possibly at the cost of some performance in naive implementations. A one-byte opcode also limits the size of the instruction set. Not assuming data alignment means that immediate data larger than a byte must be constructed from bytes at run time on many machines.

3.11.1 Types and the Java Virtual Machine

Most of the instructions in the Java virtual machine instruction set encode type information about the operations they perform. For instance, the *iload* instruction loads the contents of a local variable, which must be an `int`, onto the operand stack. The *fload* instruction does the same with a `float` value. The two instructions may have identical implementations, but have distinct opcodes.

For the majority of typed instructions, the instruction type is represented explicitly in the opcode mnemonic by a letter: *i* for an `int` operation, *l* for `long`, *s* for `short`, *b* for `byte`, *c* for `char`, *f* for `float`, *d* for `double`, and *a* for `reference`. Some instructions for which the type is unambiguous do not have a type letter in their mnemonic. For instance, *arraylength* always operates on an object that is an array. Some instructions, such as *goto*, an unconditional control transfer, do not operate on typed operands.

Given the Java virtual machine's one-byte opcode size, encoding types into opcodes places pressure on the design of its instruction set. If each typed instruction supported all of the Java virtual machine's runtime data types, there would be more instructions than could be represented in a byte. Instead, the instruction set of the Java virtual machine provides a reduced level of type support for certain operations. In other words, the instruction set is intentionally not orthogonal. Separate instructions can be used to convert between unsupported and supported data types as necessary.

[Table 3.2](#) summarizes the type support in the instruction set of the Java virtual machine. A specific instruction, with type information, is built by replacing the *T* in the instruction template in the opcode column by the letter in the type column. If the type column for some instruction template and type is blank, then no instruction exists supporting that type of operation. For instance, there is a load instruction for type `int`, *iload*, but there is no load instruction for type `byte`.

Note that most instructions in [Table 3.2](#) do not have forms for the integral types `byte`, `char`, and `short`. None have forms for the `boolean` type. Compilers encode loads of literal values of types `byte` and `short` using Java virtual machine instructions that sign-extend those values to values of

type `int` at compile time or run time. Loads of literal values of types `boolean` and `char` are encoded using instructions that zero-extend the literal to a value of type `int` at compile time or run time. Likewise, loads from arrays of values of type `boolean`, `byte`, `short`, and `char` are encoded using Java virtual machine instructions that sign-extend or zero-extend the values to values of type `int`. Thus, most operations on values of actual types `boolean`, `byte`, `char`, and `short` are correctly performed by instructions operating on values of computational type `int`.

<i>opcode</i>	<i>byte</i>	<i>short</i>	<i>int</i>	<i>long</i>	<i>float</i>	<i>double</i>	<i>char</i>	<i>reference</i>
<i>Tipush</i>	<i>bipush</i>	<i>sipush</i>						
<i>Tconst</i>			<i>iconst</i>	<i>lconst</i>	<i>fconst</i>	<i>dconst</i>		<i>aconst</i>
<i>Tload</i>			<i>iload</i>	<i>lload</i>	<i>float</i>	<i>dload</i>		<i>aload</i>
<i>Tstore</i>			<i>istore</i>	<i>lstore</i>	<i>fstore</i>	<i>dstore</i>		<i>astore</i>
<i>Tinc</i>			<i>iinc</i>					
<i>Taload</i>	<i>baload</i>	<i>saload</i>	<i>iaload</i>	<i>laload</i>	<i>faload</i>	<i>daload</i>	<i>caload</i>	<i>aaload</i>
<i>Tastore</i>	<i>bastore</i>	<i>sastore</i>	<i>iastore</i>	<i>lastore</i>	<i>fastore</i>	<i>dastore</i>	<i>castore</i>	<i>aastore</i>
<i>Tadd</i>			<i>iadd</i>	<i>ladd</i>	<i>fadd</i>	<i>dadd</i>		
<i>Tsub</i>			<i>isub</i>	<i>lsub</i>	<i>fsub</i>	<i>dsub</i>		
<i>Tmul</i>			<i>imul</i>	<i>lmul</i>	<i>fmul</i>	<i>dmul</i>		
<i>Tdiv</i>			<i>idiv</i>	<i>ldiv</i>	<i>fdiv</i>	<i>ddiv</i>		
<i>Trem</i>			<i>irem</i>	<i>lrem</i>	<i>frem</i>	<i>drem</i>		
<i>Tneg</i>			<i>ineg</i>	<i>lneg</i>	<i>fneg</i>	<i>dneg</i>		
<i>Tshl</i>			<i>ishl</i>	<i>lshl</i>				
<i>Tshr</i>			<i>ishr</i>	<i>lshr</i>				
<i>Tushr</i>			<i>iushr</i>	<i>lushr</i>				
<i>Tand</i>			<i>iand</i>	<i>land</i>				
<i>Tor</i>			<i>ior</i>	<i>lor</i>				
<i>Txor</i>			<i>ixor</i>	<i>lxor</i>				
<i>i2T</i>	<i>i2b</i>	<i>i2s</i>		<i>i2l</i>	<i>i2f</i>	<i>i2d</i>		
<i>l2T</i>			<i>l2i</i>		<i>l2f</i>	<i>l2d</i>		
<i>f2T</i>			<i>f2i</i>	<i>f2l</i>		<i>f2d</i>		
<i>d2T</i>			<i>d2i</i>	<i>d2l</i>	<i>d2f</i>			
<i>Tcmp</i>				<i>lcmp</i>				
<i>Tcmpl</i>					<i>fcmpl</i>	<i>dcmpl</i>		
<i>Tcmpg</i>					<i>fcmpg</i>	<i>dcmpg</i>		
<i>if_TcmpOP</i>			<i>if_icmpOP</i>					<i>if_acmpOP</i>

*Treturn**ireturn**lreturn**freturn**dreturn**areturn*

The mapping between Java virtual machine actual types and Java virtual machine computational types is summarized by [Table 3.3](#).

<i>Actual Type</i>	<i>Computational Type</i>	<i>Category</i>
boolean	int	category 1
byte	int	category 1
char	int	category 1
short	int	category 1
int	int	category 1
float	float	category 1
reference	reference	category 1
returnAddress	returnAddress	category 1
long	long	category 2
double	double	category 2

Certain Java virtual machine instructions such as *pop* and *swap* operate on the operand stack without regard to type; however, such instructions are constrained to use only on values of certain categories of computational types, also given in [Table 3.3](#).

The remainder of this chapter summarizes the Java virtual machine instruction set.

3.11.2 Load and Store Instructions

The load and store instructions transfer values between the local variables ([§3.6.1](#)) and the operand stack ([§3.6.2](#)) of a Java virtual machine frame ([§3.6](#)):

- Load a local variable onto the operand stack: *iload*, *iload_<n>*, *lload*, *lload_<n>*, *fload*, *fload_<n>*, *dload*, *dload_<n>*, *aload*, *aload_<n>*.
- Store a value from the operand stack into a local variable: *istore*, *istore_<n>*, *lstore*, *lstore_<n>*, *fstore*, *fstore_<n>*, *dstore*, *dstore_<n>*, *astore*, *astore_<n>*.
- Load a constant onto the operand stack: *bipush*, *sipush*, *ldc*, *ldc_w*, *ldc2_w*, *aconst_null*, *iconst_m1*, *iconst_<i>*, *lconst_<l>*, *fconst_<f>*, *dconst_<d>*.
- Gain access to more local variables using a wider index, or to a larger immediate operand: *wide*.

Instructions that access fields of objects and elements of arrays ([§3.11.5](#)) also transfer data to and from the operand stack.

Instruction mnemonics shown above with trailing letters between angle brackets (for instance, *iload_<n>*) denote families of instructions (with members *iload_0*, *iload_1*, *iload_2*, and *iload_3* in the case of *iload_<n>*). Such families of instructions are specializations of an additional generic instruction (*iload*) that takes one operand. For the specialized instructions, the operand is implicit and does not need to be stored or fetched. The semantics are otherwise the same (*iload_0* means the same thing as *iload* with the operand 0). The letter between the angle brackets specifies the type of the implicit operand for that family of instructions: for *<n>*, a nonnegative integer; for *<i>*, an `int`; for *<l>*, a `long`; for *<f>*, a `float`; and for *<d>*, a `double`. Forms for type `int` are used in many cases to perform operations on values of type `byte`, `char`, and `short` ([§3.11.1](#)).

This notation for instruction families is used throughout *The Java™ Virtual Machine Specification*.

3.11.3 Arithmetic Instructions

The arithmetic instructions compute a result that is typically a function of two values on the operand stack, pushing the result back on the operand stack. There are two main kinds of arithmetic instructions: those operating on integer values and those operating on floating-point values. Within each of these kinds, the arithmetic instructions are specialized to Java virtual machine numeric types. There is no direct support for integer arithmetic on values of the `byte`, `short`, and `char` types ([§3.11.1](#)), or for values of the `boolean` type; those operations are handled by instructions operating on type `int`. Integer and floating-point instructions also differ in their behavior on overflow and divide-by-zero. The arithmetic instructions are as follows:

- Add: *iadd*, *ladd*, *fadd*, *dadd*.
- Subtract: *isub*, *lsub*, *fsub*, *dsub*.
- Multiply: *imul*, *lmul*, *fmul*, *dmul*.

- Divide: *idiv*, *ldiv*, *fdiv*, *ddiv*.
- Remainder: *irem*, *lrem*, *frem*, *drem*.
- Negate: *ineg*, *lneg*, *fneg*, *dneg*.
- Shift: *ishl*, *ishr*, *iushr*, *lshl*, *lshr*, *lushr*.
- Bitwise OR: *ior*, *lor*.
- Bitwise AND: *iand*, *land*.
- Bitwise exclusive OR: *ixor*, *lxor*.
- Local variable increment: *iinc*.
- Comparison: *dcmprg*, *dcmpl*, *fcmpg*, *fcmpl*, *lcmp*.

The semantics of the Java programming language operators on integer and floating-point values ([§2.4.2](#), [§2.4.4](#)) are directly supported by the semantics of the Java virtual machine instruction set.

The Java virtual machine does not indicate overflow during operations on integer data types. The only integer operations that can throw an exception are the integer divide instructions (*idiv* and *ldiv*) and the integer remainder instructions (*irem* and *lrem*), which throw an `ArithmeticException` if the divisor is zero.

Java virtual machine operations on floating-point numbers behave as specified in IEEE 754. In particular, the Java virtual machine requires full support of IEEE 754 *denormalized* floating-point numbers and *gradual underflow*, which make it easier to prove desirable properties of particular numerical algorithms.

The Java virtual machine requires that floating-point arithmetic behave as if every floating-point operator rounded its floating-point result to the result precision. *Inexact* results must be rounded to the representable value nearest to the infinitely precise result; if the two nearest representable values are equally near, the one having a least significant bit of zero is chosen. This is the IEEE 754 standard's default rounding mode, known as *round to nearest* mode.

The Java virtual machine uses the IEEE 754 *round towards zero* mode when converting a floating-point value to an integer. This results in the number being truncated; any bits of the significand that represent the fractional part of the operand value are discarded. Round towards zero mode chooses as its result the type's value closest to, but no greater in magnitude than, the infinitely precise result.

The Java virtual machine's floating-point operators do not throw runtime exceptions (not to be confused with IEEE 754 floating-point exceptions). An operation that overflows produces a signed

infinity, an operation that underflows produces a denormalized value or a signed zero, and an operation that has no mathematically definite result produces NaN. All numeric operations with NaN as an operand produce NaN as a result.

Comparisons on values of type `long` (*lcmp*) perform a signed comparison. Comparisons on values of floating-point types (*dcmpg*, *dcmpl*, *fcmpg*, *fcmpl*) are performed using IEEE 754 nonsignaling comparisons.

3.11.4 Type Conversion Instructions

The type conversion instructions allow conversion between Java virtual machine numeric types. These may be used to implement explicit conversions in user code or to mitigate the lack of orthogonality in the instruction set of the Java virtual machine.

The Java virtual machine directly supports the following widening numeric conversions:

- `int` to `long`, `float`, or `double`
- `long` to `float` or `double`
- `float` to `double`

The widening numeric conversion instructions are *i2l*, *i2f*, *i2d*, *l2f*, *l2d*, and *f2d*. The mnemonics for these opcodes are straightforward given the naming conventions for typed instructions and the punning use of 2 to mean "to." For instance, the *i2d* instruction converts an `int` value to a `double`. Widening numeric conversions do not lose information about the overall magnitude of a numeric value. Indeed, conversions widening from `int` to `long` and `int` to `double` do not lose any information at all; the numeric value is preserved exactly. Conversions widening from `float` to `double` that are FP-strict ([§3.8.2](#)) also preserve the numeric value exactly; however, such conversions that are not FP-strict may lose information about the overall magnitude of the converted value.

Conversion of an `int` or a `long` value to `float`, or of a `long` value to `double`, may lose *precision*, that is, may lose some of the least significant bits of the value; the resulting floating-point value is a correctly rounded version of the integer value, using IEEE 754 round to nearest mode.

A widening numeric conversion of an `int` to a `long` simply sign-extends the two's-complement representation of the `int` value to fill the wider format. A widening numeric conversion of a `char` to an integral type zero-extends the representation of the `char` value to fill the wider format.

Despite the fact that loss of precision may occur, widening numeric conversions never cause the Java virtual machine to throw a runtime exception (not to be confused with an IEEE 754 floating-point exception).

Note that widening numeric conversions do not exist from integral types `byte`, `char`, and `short` to type `int`. As noted in [§3.11.1](#), values of type `byte`, `char`, and `short` are internally widened to type `int`, making these conversions implicit.

The Java virtual machine also directly supports the following narrowing numeric conversions:

- `int` to `byte`, `short`, or `char`
- `long` to `int`
- `float` to `int` or `long`
- `double` to `int`, `long`, or `float`

The narrowing numeric conversion instructions are *i2b*, *i2c*, *i2s*, *l2i*, *f2i*, *f2l*, *d2i*, *d2l*, and *d2f*. A narrowing numeric conversion can result in a value of different sign, a different order of magnitude, or both; it may thereby lose precision.

A narrowing numeric conversion of an `int` or `long` to an integral type `T` simply discards all but the `N` lowest-order bits, where `N` is the number of bits used to represent type `T`. This may cause the resulting value not to have the same sign as the input value.

In a narrowing numeric conversion of a floating-point value to an integral type `T`, where `T` is either `int` or `long`, the floating-point value is converted as follows:

- If the floating-point value is NaN, the result of the conversion is an `int` or `long` 0.
- Otherwise, if the floating-point value is not an infinity, the floating-point value is rounded to an integer value `V` using IEEE 754 round towards zero mode. There are two cases:
 - If `T` is `long` and this integer value can be represented as a `long`, then the result is the `long` value `V`.
 - If `T` is of type `int` and this integer value can be represented as an `int`, then the result is the `int` value `V`.
- Otherwise:
 - Either the value must be too small (a negative value of large magnitude or negative infinity), and the result is the smallest representable value of type `int` or `long`.
 - Or the value must be too large (a positive value of large magnitude or positive infinity), and the result is the largest representable value of type `int` or `long`.

A narrowing numeric conversion from `double` to `float` behaves in accordance with IEEE 754. The result is correctly rounded using IEEE 754 round to nearest mode. A value too small to be represented as a `float` is converted to a positive or negative zero of type `float`; a value too large to be represented as a `float` is converted to a positive or negative infinity. A `double` NaN is always converted to a `float` NaN.

Despite the fact that overflow, underflow, or loss of precision may occur, narrowing conversions among numeric types never cause the Java virtual machine to throw a runtime exception (not to be confused with an IEEE 754 floating-point exception).

3.11.5 Object Creation and Manipulation

Although both class instances and arrays are objects, the Java virtual machine creates and manipulates class instances and arrays using distinct sets of instructions:

- Create a new class instance: *new*.
- Create a new array: *newarray*, *anewarray*, *multianewarray*.
- Access fields of classes (`static` fields, known as class variables) and fields of class instances (non-`static` fields, known as instance variables): *getfield*, *putfield*, *getstatic*, *putstatic*.
- Load an array component onto the operand stack: *baload*, *caload*, *saload*, *iaload*, *laload*, *faload*, *daload*, *aaload*.
- Store a value from the operand stack as an array component: *bastore*, *castore*, *sastore*, *iastore*, *lastore*, *fastore*, *dastore*, *aastore*.
- Get the length of array: *arraylength*.
- Check properties of class instances or arrays: *instanceof*, *checkcast*.

3.11.6 Operand Stack Management Instructions

A number of instructions are provided for the direct manipulation of the operand stack: *pop*, *pop2*, *dup*, *dup2*, *dup_x1*, *dup2_x1*, *dup_x2*, *dup2_x2*, *swap*.

3.11.7 Control Transfer Instructions

The control transfer instructions conditionally or unconditionally cause the Java virtual machine to continue execution with an instruction other than the one following the control transfer instruction. They are:

- Conditional branch: *ifeq*, *iflt*, *ifle*, *ifne*, *ifgt*, *ifge*, *ifnull*, *ifnonnull*, *if_icmpeq*, *if_icmpne*, *if_icmplt*, *if_icmpgt*, *if_icmple*, *if_icmpge*, *if_acmpeq*, *if_acmpne*.
- Compound conditional branch: *tableswitch*, *lookupswitch*.
- Unconditional branch: *goto*, *goto_w*, *jsr*, *jsr_w*, *ret*.

The Java virtual machine has distinct sets of instructions that conditionally branch on comparison with data of `int` and `reference` types. It also has distinct conditional branch instructions that test for the null reference and thus is not required to specify a concrete value for `null` ([§3.4](#)).

Conditional branches on comparisons between data of types `boolean`, `byte`, `char`, and `short` are performed using `int` comparison instructions ([§3.11.1](#)). A conditional branch on a comparison between data of types `long`, `float`, or `double` is initiated using an instruction that compares the data and produces an `int` result of the comparison ([§3.11.3](#)). A subsequent `int` comparison instruction tests this result and effects the conditional branch. Because of its emphasis on `int` comparisons, the Java virtual machine provides a rich complement of conditional branch instructions for type `int`.

All `int` conditional control transfer instructions perform signed comparisons.

3.11.8 Method Invocation and Return Instructions

The following four instructions invoke methods:

- *invokevirtual* invokes an instance method of an object, dispatching on the (virtual) type of the object. This is the normal method dispatch in the Java programming language.
- *invokeinterface* invokes a method that is implemented by an interface, searching the methods implemented by the particular runtime object to find the appropriate method.
- *invokespecial* invokes an instance method requiring special handling, whether an instance initialization method ([§3.9](#)), a `private` method, or a superclass method.
- *invokestatic* invokes a class (`static`) method in a named class.

The method return instructions, which are distinguished by return type, are *ireturn* (used to return values of type `boolean`, `byte`, `char`, `short`, or `int`), *lreturn*, *freturn*, *dreturn*, and *areturn*. In addition, the *return* instruction is used to return from methods declared to be `void`, instance initialization methods, and class or interface initialization methods.

3.11.9 Throwing Exceptions

An exception is thrown programmatically using the *athrow* instruction. Exceptions can also be thrown by various Java virtual machine instructions if they detect an abnormal condition.

3.11.10 Implementing `finally`

The implementation of the `finally` keyword uses the *jsr*, *jsr_w*, and *ret* instructions. See [Section 4.9.6, "Exceptions and finally,"](#) and [Section 7.13, "Compiling finally."](#)

3.11.11 Synchronization

The Java virtual machine supports synchronization of both methods and sequences of instructions within a method using a single synchronization construct: the *monitor* .

Method-level synchronization is handled as part of method invocation and return (see [Section 3.11.8, "Method Invocation and Return Instructions"](#)).

Synchronization of sequences of instructions is typically used to encode the synchronized blocks of the Java programming language. The Java virtual machine supplies the *monitorenter* and *monitorexit* instructions to support such constructs.

Proper implementation of synchronized blocks requires cooperation from a compiler targeting the Java virtual machine. The compiler must ensure that at any method invocation completion a *monitorexit* instruction will have been executed for each *monitorenter* instruction executed since the method invocation. This must be the case whether the method invocation completes normally ([§3.6.4](#)) or abruptly ([§3.6.5](#)).

The compiler enforces proper pairing of *monitorenter* and *monitorexit* instructions on abrupt method invocation completion by generating exception handlers ([§3.10](#)) that will match any exception and whose associated code executes the necessary *monitorexit* instructions ([§7.14](#)).

3.12 Class Libraries

The Java virtual machine must provide sufficient support for the implementation of the class libraries of the associated platform. Some of the classes in these libraries cannot be implemented without the cooperation of the Java virtual machine.

Classes that might require special support from the Java virtual machine include those that support:

- Reflection, such as the classes in the package `java.lang.reflect` and the class `Class`.
- Loading and creation of a class or interface. The most obvious example is the class

ClassLoader.

- Linking and initialization of a class or interface. The example classes cited above fall into this category as well.
- Security, such as the classes in the package `java.security` and other classes such as `SecurityManager`.
- Multithreading, such as the class `Thread`.
- Weak references, such as the classes in the package `java.lang.ref`.^{[9](#)}

The list above is meant to be illustrative rather than comprehensive. An exhaustive list of these classes or of the functionality they provide is beyond the scope of this book. See the specifications of the Java and Java 2 platform class libraries for details.

3.13 Public Design, Private Implementation

Thus far this book has sketched the public view of the Java virtual machine: the `class` file format and the instruction set. These components are vital to the hardware-, operating system-, and implementation-independence of the Java virtual machine. The implementor may prefer to think of them as a means to securely communicate fragments of programs between hosts each implementing the Java or Java 2 platform, rather than as a blueprint to be followed exactly.

It is important to understand where the line between the public design and the private implementation lies. A Java virtual machine implementation must be able to read `class` files and must exactly implement the semantics of the Java virtual machine code therein. One way of doing this is to take this document as a specification and to implement that specification literally. But it is also perfectly feasible and desirable for the implementor to modify or optimize the implementation within the constraints of this specification. So long as the `class` file format can be read and the semantics of its code are maintained, the implementor may implement these semantics in any way. What is "under the hood" is the implementor's business, as long as the correct external interface is carefully maintained.^{[10](#)}

The implementor can use this flexibility to tailor Java virtual machine implementations for high performance, low memory use, or portability. What makes sense in a given implementation depends on the goals of that implementation. The range of implementation options includes the following:

- Translating Java virtual machine code at load time or during execution into the instruction set of another virtual machine.
- Translating Java virtual machine code at load time or during execution into the native instruction set of the host CPU (sometimes referred to as *just-in-time*, or *JIT*, code generation).

The existence of a precisely defined virtual machine and object file format need not significantly restrict the creativity of the implementor. The Java virtual machine is designed to support many different implementations, providing new and interesting solutions while retaining compatibility between implementations.

¹ The first edition of *The JavaTM Virtual Machine Specification* did not consider `boolean` to be a Java virtual machine type. However, `boolean` values do have limited support in the Java virtual machine. This second edition clarifies the issue by treating `boolean` as a type.

² In Sun's JDK releases 1.0 and 1.1, and the Java 2 SDK, Standard Edition, v1.2, `boolean` arrays in the Java programming language are encoded as Java virtual machine `byte` arrays, using 8 bits per `boolean` element.

³ In the first edition of this specification, the Java virtual machine stack was known as the *Java stack*.

⁴ In Sun's implementations of the Java virtual machine in JDK releases 1.0.2 and 1.1, and the Java 2 SDK, Standard Edition, v1.2, Java virtual machine stacks are discontinuous and are independently expanded as required by the computation. Those implementations do not free memory allocated for a Java virtual machine stack until the associated thread terminates. Expansion is subject to a size limit for any one stack. The Java virtual machine stack size limit may be set on virtual machine start-up using the `"-oss"` flag. The Java virtual machine stack size limit can be used to limit memory consumption or to catch runaway recursions.

⁵ Sun's implementations of the Java virtual machine in JDK releases 1.0.2 and 1.1, and the Java 2 SDK, Standard Edition, v1.2, dynamically expand the heap as required by the computation, but never contract the heap. The initial and maximum sizes may be specified on virtual machine start-up using the `"-ms"` and `"-mx"` flags, respectively.

⁶ Sun's implementation of the Java virtual machine in JDK release 1.0.2 dynamically expands the method area as required by the computation, but never contracts the method area. The Java virtual machine implementations in Sun's JDK release 1.1 and the Java 2 SDK, Standard Edition, v1.2 garbage collect the method area. In neither case is user control over the initial, minimum, or maximum size of the method area provided.

⁷ Sun's implementations of the Java virtual machine in JDK releases 1.0.2 and 1.1, and the Java 2 SDK, Standard Edition, v1.2, allocate fixed-size native method stacks of a single size. The size of the native method stacks may be set on virtual machine start-up using the `"-ss"` flag. The native method stack size limit can be used to limit memory consumption or to catch runaway recursions in `native` methods. Sun's implementations do *not* check for native method stack overflow.

⁸ In some of Sun's implementations of the Java virtual machine, a reference to a class instance is a pointer to a *handle* that is itself a pair of pointers: one to a table containing the methods of the object

and a pointer to the `Class` object that represents the type of the object, and the other to the memory allocated from the heap for the object data.

⁹ Weak references were introduced in the Java 2 platform, v1.2.

¹⁰ There are some exceptions: debuggers, profilers, and just-in-time code generators can each require access to elements of the Java virtual machine that are normally considered to be "under the hood." Where appropriate, Sun is working with other Java virtual machine implementors and tools vendors to develop common interfaces to the Java virtual machine for use by such tools, and to promote those interfaces across the industry. Information on publicly available low-level interfaces to the Java virtual machine will be made available at <http://java.sun.com>.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The JavaTM Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections through our [feedback form](#)

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

CHAPTER 6

The Java Virtual Machine Instruction Set

A Java virtual machine instruction consists of an opcode specifying the operation to be performed, followed by zero or more operands embodying values to be operated upon. This chapter gives details about the format of each Java virtual machine instruction and the operation it performs.

6.1 Assumptions: The Meaning of "Must"

The description of each instruction is always given in the context of Java virtual machine code that satisfies the static and structural constraints of [Chapter 4](#), "[The `class` File Format](#)." In the description of individual Java virtual machine instructions, we frequently state that some situation "must" or "must not" be the case: "The *value2* must be of type `int`." The constraints of [Chapter 4](#) guarantee that all such expectations will in fact be met. If some constraint (a "must" or "must not") in an instruction description is not satisfied at run time, the behavior of the Java virtual machine is undefined.

The Java virtual machine checks that Java virtual machine code satisfies the static and structural constraints at link time using a `class` file verifier (see [Section 4.9](#), "[Verification of `class` Files](#)"). Thus, a Java virtual machine will only attempt to execute code from valid `class` files. Performing verification at link time is attractive in that the checks are performed just once, substantially reducing the amount of work that must be done at run time. Other implementation strategies are possible, provided that they comply with *The JavaTM Language Specification* and *The JavaTM Virtual Machine Specification*.

6.2 Reserved Opcodes

In addition to the opcodes of the instructions specified later in this chapter, which are used in `class` files (see [Chapter 4](#), "[The `class` File Format](#)"), three opcodes are reserved for internal use by a Java

virtual machine implementation. If Sun extends the instruction set of the Java virtual machine in the future, these reserved opcodes are guaranteed not to be used.

Two of the reserved opcodes, numbers 254 (0xfe) and 255 (0xff), have the mnemonics *impdep1* and *impdep2*, respectively. These instructions are intended to provide "back doors" or traps to implementation-specific functionality implemented in software and hardware, respectively. The third reserved opcode, number 202 (0xca), has the mnemonic *breakpoint* and is intended to be used by debuggers to implement breakpoints.

Although these opcodes have been reserved, they may be used only inside a Java virtual machine implementation. They cannot appear in valid `class` files. Tools such as debuggers or JIT code generators ([§3.13](#)) that might directly interact with Java virtual machine code that has been already loaded and executed may encounter these opcodes. Such tools should attempt to behave gracefully if they encounter any of these reserved instructions.

6.3 Virtual Machine Errors

A Java virtual machine implementation throws an object that is an instance of a subclass of the class `VirtualMachineError` when an internal error or resource limitation prevents it from correctly implementing the Java programming language. The Java virtual machine specification cannot predict where resource limitations or internal errors may be encountered and does not mandate precisely when they can be reported. Thus, any of the virtual machine errors listed as subclasses of `VirtualMachineError` in [Section 2.16.4](#) may be thrown at any time during the operation of the Java virtual machine.

6.4 Format of Instruction Descriptions

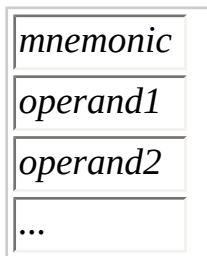
Java virtual machine instructions are represented in this chapter by entries of the form shown in [Figure 6.1](#), in alphabetical order and each beginning on a new page. For example:

mnemonic

Operation

Short description of the instruction

Format



Forms

mnemonic = opcode

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *value3*

Description

A longer description detailing constraints on operand stack contents or constant pool entries, the operation performed, the type of the results, etc.

Linking Exceptions

If any linking exceptions may be thrown by the execution of this instruction, they are set off one to a line, in the order in which they must be thrown.

Runtime Exceptions

If any runtime exceptions can be thrown by the execution of an instruction, they are set off one to a line, in the order in which they must be thrown.

Other than the linking and runtime exceptions, if any, listed for an instruction, that instruction must not throw any runtime exceptions except for instances of `VirtualMachineError` or its subclasses.

Notes

Comments not strictly part of the specification of an instruction are set aside as notes at the end of the description.

Each cell in the instruction format diagram represents a single 8-bit byte. The instruction's *mnemonic*

is its name. Its opcode is its numeric representation and is given in both decimal and hexadecimal forms. Only the numeric representation is actually present in the Java virtual machine code in a `class` file.

Keep in mind that there are "operands" generated at compile time and embedded within Java virtual machine instructions, as well as "operands" calculated at run time and supplied on the operand stack. Although they are supplied from several different areas, all these operands represent the same thing: values to be operated upon by the Java virtual machine instruction being executed. By implicitly taking many of its operands from its operand stack, rather than representing them explicitly in its compiled code as additional operand bytes, register numbers, etc., the Java virtual machine's code stays compact.

Some instructions are presented as members of a family of related instructions sharing a single description, format, and operand stack diagram. As such, a family of instructions includes several opcodes and opcode mnemonics; only the family mnemonic appears in the instruction format diagram, and a separate forms line lists all member mnemonics and opcodes. For example, the forms line for the *lconst_<l>* family of instructions, giving mnemonic and opcode information for the two instructions in that family (*lconst_0* and *lconst_1*), is

Forms

lconst_0 = 9 (0x9)

lconst_1 = 10 (0xa)

In the description of the Java virtual machine instructions, the effect of an instruction's execution on the operand stack ([§3.6.2](#)) of the current frame ([§3.6](#)) is represented textually, with the stack growing from left to right and each value represented separately. Thus,

Operand Stack

..., value1, value2 $\Rightarrow$ *..., result*

shows an operation that begins by having *value2* on top of the operand stack with *value1* just beneath it. As a result of the execution of the instruction, *value1* and *value2* are popped from the operand stack and replaced by *result* value, which has been calculated by the instruction. The remainder of the operand stack, represented by an ellipsis (...), is unaffected by the instruction's execution.

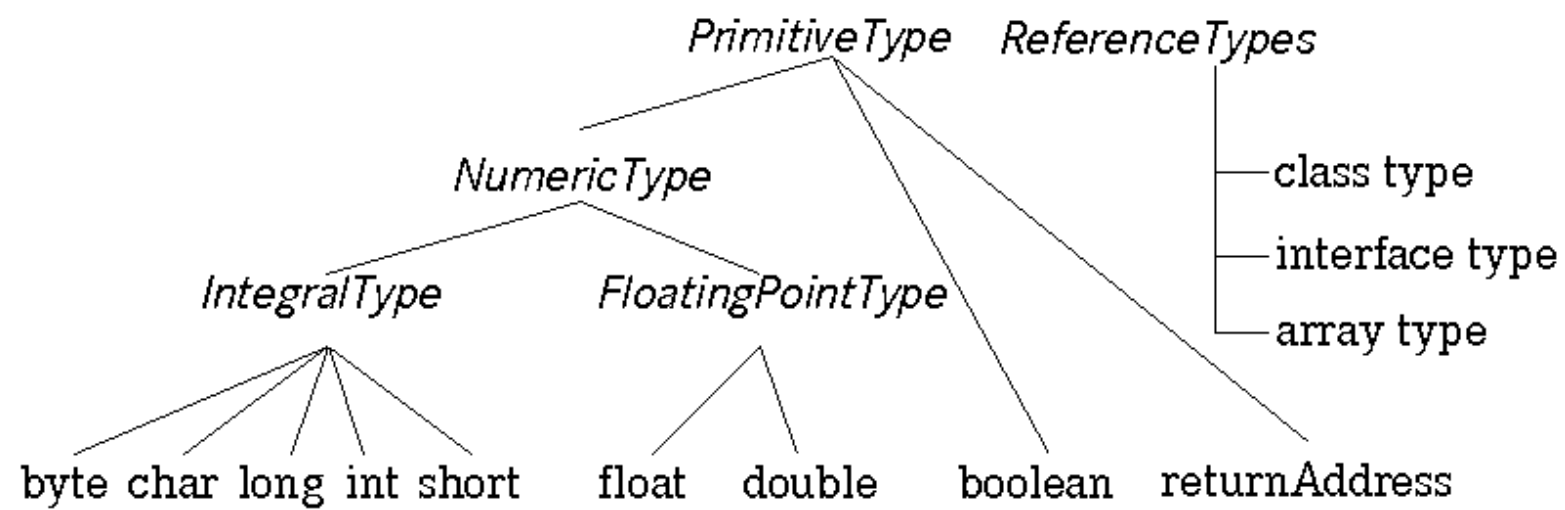
Values of types `long` and `double` are represented by a single entry on the operand stack.^{[1](#)}

¹ Note that, in the first edition of this specification, values on the operand stack of types `long` and `double` were each represented in the stack diagram by two entries.

The JavaTM Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections through our [feedback form](#)



IEEE Std 754-1985 IEEE Standard for Binary Floating-Point Arithmetic - Description

Content

- **1. Scope**
 - 1.1 Implementation Objectives
 - 1.2 Inclusions
 - 1.3 Exclusions
- **2. Definitions**
- **3. Formats**
 - 3.1 Sets of Values
 - 3.2 Basic Formats
 - 3.3 Extended Formats
 - 3.4 Combinations of Formats
- **4. Rounding**
 - 4.1 Round to Nearest
 - 4.2 Directed Roundings
 - 4.3 Rounding Precision
- **5. Operations**
 - 5.1 Arithmetic
 - 5.2 Square Root
 - 5.3 Floating-Point Format Conversions
 - 5.4 Conversion Between Floating-Point and Integer Formats
 - 5.5 Round Floating-Point Number to Integer Value
 - 5.6 Binary Decimal Conversion
 - 5.7 Comparison
- **6. Infinity, NaNs, and Signed Zero**
 - 6.1 Infinity Arithmetic
 - 6.2 Operations with NaNs
 - 6.3 The Sign Bit
- **7. Exceptions**
 - 7.1 Invalid Operation
 - 7.2 Division by Zero

- 7.3 Overflow
- 7.4 Underflow
- 7.5 Inexact

- **8. Traps**

- 8.1 Trap Handler
- 8.2 Precedence

- **Annex A Recommended Functions and Predicates**

links: [[Ordering Information](#)] - [[Catalog](#)] - [[Standard Status](#)]

[Copyright © 2003 IEEE](#)

(m.v.rodriquez@ieee.org)

URL:

(Modified: Tue Nov 4 03:48:17 2003)

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

ret

Operation

Return from subroutine

Format



Forms

ret = 169 (0xa9)

Operand Stack

No change

Description

The *index* is an unsigned byte between 0 and 255, inclusive. *The local variable at index in the current frame* ([§3.6](#)) *must contain a value of type* `returnAddress`. The contents of the local variable are written into the Java virtual machine's pc register, and execution continues there.

Notes

The *ret* instruction is used with *jsr* or *jsr_w* instructions in the implementation of the `finally` clauses of the Java programming language (see [Section 7.13, "Compiling finally"](#)). Note that *jsr* pushes the address onto the operand stack and *ret* gets it out of a local variable. This asymmetry is intentional.

The *ret* instruction should not be confused with the *return* instruction. A *return* instruction returns control from a method to its invoker, without passing any value

back to the invoker.

The *ret* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

return

Operation

Return `void` from method

Format



Forms

return = 177 (0xb1)

Operand Stack

... $\Rightarrow$ [empty]

Description

The current method must have return type `void`. If the current method is a *synchronized* method, the monitor acquired or reentered on invocation of the method is released or exited (respectively) as if by execution of a *monitorexit* instruction. If no exception is thrown, any values on the operand stack of the current frame ([§3.6](#)) are discarded.

The interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

Runtime Exceptions

If the current method is a *synchronized* method and the current thread is not the owner of the monitor acquired or reentered on invocation of the method, *return* throws an `IllegalMonitorStateException`. This can happen, for example, if a *synchronized* method contains a *monitorexit* instruction, but no *monitorenter*

instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in [Section 8.13](#) and if the first of those rules is violated during invocation of the current method, then *return* throws an `IllegalMonitorStateException`.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The JavaTM Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections to jvm@java.sun.com

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

aaload

Operation

Load reference from array

Format

<i>aaload</i>

Forms

aaload = 50 (0x32)

Operand Stack

..., *arrayref*, *index* $\Rightarrow$..., *value*

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `reference`. The *index* must be of type `int`. Both *arrayref* and *index* are popped from the operand stack. The `reference` *value* in the component of the array at *index* is retrieved and pushed onto the operand stack.

Runtime Exceptions

If *arrayref* is `null`, *aaload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *aaload* instruction throws an `ArrayIndexOutOfBoundsException`.

aastore

Operation

Store into reference array

Format

<i>aastore</i>

Forms

aastore = 83 (0x53)

Operand Stack

..., *arrayref*, *index*, *value* $\Rightarrow$...

Description

The *arrayref* must be of type **reference** and must refer to an array whose components are of type **reference**. The *index* must be of type **int** and *value* must be of type **reference**. The *arrayref*, *index*, and *value* are popped from the operand stack. The **reference value** is stored as the component of the array at *index*.

The type of *value* must be assignment compatible ([§2.6.7](#)) with the type of the components of the array referenced by *arrayref*. Assignment of a value of reference type S (source) to a variable of reference type T (target) is allowed only when the type S supports all the operations defined on type T. The detailed rules follow:

- If S is a class type, then:
 - If T is a class type, then S must be the same class ([§2.8.1](#)) as T, or S must be a subclass of T;
 - If T is an interface type, S must implement ([§2.13](#)) interface T.
- If S is an interface type, then:
 - If T is a class type, then T must be **Object** ([§2.4.7](#)).
 - If T is an interface type, then T must be the same interface as S or a superinterface of S ([§2.13.2](#)).

- If *S* is an array type, namely, the type *SC*[], that is, an array of components of type *SC*, then:
 - If *T* is a class type, then *T* must be `Object` (§2.4.7).
 - If *T* is an array type *TC*[], that is, an array of components of type *TC*, then one of the following must be true:
 - *TC* and *SC* are the same primitive type (§2.4.1).
 - *TC* and *SC* are reference types (§2.4.6), and type *SC* is assignable to *TC* by these runtime rules.
 - If *T* is an interface type, *T* must be one of the interfaces implemented by arrays (§2.15).

Runtime Exceptions

If *arrayref* is `null`, *aastore* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *aastore* instruction throws an `ArrayIndexOutOfBoundsException`.

Otherwise, if *arrayref* is not `null` and the actual type of *value* is not assignment compatible (§2.6.7) with the actual type of the components of the array, *aastore* throws an `ArrayStoreException`.

`aconst_null`

Operation

Push `null`

Format

<code>aconst_null</code>

Forms

`aconst_null` = 1 (0x1)

Operand Stack

... $\Rightarrow$..., *null*

Description

Push the `null` object reference onto the operand stack.

Notes

The Java virtual machine does not mandate a concrete value for `null`.

aload

Operation

Load reference from local variable

Format

<i>aload</i>
<i>index</i>

Forms

aload = 25 (0x19)

Operand Stack

... $\Rightarrow$..., *objectref*

Description

The *index* is an unsigned byte that must be an index into the local variable array of the current frame ([§3.6](#)). The local variable at *index* must contain a `reference`. The *objectref* in the local variable at *index* is pushed onto the operand stack.

Notes

The *aload* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction is intentional.

The *aload* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

aload_<n>

Operation

Load reference from local variable

Format

<i>aload_<n></i>

Forms

aload_0 = 42 (0x2a) *aload_1* = 43 (0x2b) *aload_2* = 44 (0x2c) *aload_3* = 45 (0x2d)

Operand Stack

... $\Rightarrow$..., *objectref*

Description

The <*n*> must be an index into the local variable array of the current frame ([§3.6](#)). The local variable at <*n*> must contain a **reference**. The *objectref* in the local variable at *index* is pushed onto the operand stack.

Notes

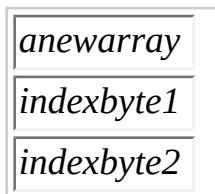
An *aload_<n>* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the corresponding *astore_<n>* instruction is intentional. Each of the *aload_<n>* instructions is the same as *aload* with an *index* of <*n*>, except that the operand <*n*> is implicit.

anewarray

Operation

Create new array of reference

Format



Forms

anewarray = 189 (0xbd)

Operand Stack

..., *count* $\Rightarrow$..., *arrayref*

Description

The *count* must be of type `int`. It is popped off the operand stack. The *count* represents the number of components of the array to be created. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class ([§3.6](#)), where the value of the index is $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$. The runtime constant pool item at that index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved ([§5.4.3.1](#)). A new array with components of that type, of length *count*, is allocated from the garbage-collected heap, and a reference *arrayref* to this new array object is pushed onto the operand stack. All components of the new array are initialized to `null`, the default value for reference types ([§2.5.1](#)).

Linking Exceptions

During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in [§5.4.3.1](#) can be thrown.

Runtime Exception

Otherwise, if *count* is less than zero, the *anewarray* instruction throws a

`NegativeArraySizeException`.

Notes

The *anewarray* instruction is used to create a single dimension of an array of object references or part of a multidimensional array.

areturn

Operation

Return **reference** from method

Format

<i>areturn</i>

Forms

areturn = 176 (0xb0)

Operand Stack

..., *objectref* $\Rightarrow$ [empty]

Description

The *objectref* must be of type **reference** and must refer to an object of a type that is assignment compatible ([§2.6.7](#)) with the type represented by the return descriptor ([§4.3.3](#)) of the current method. If the current method is a **synchronized** method, the monitor acquired or reentered on invocation of the method is released or exited (respectively) as if by execution of a *monitorexit* instruction. If no exception is thrown, *objectref* is popped from the operand stack of the current frame ([§3.6](#)) and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The interpreter then reinstates the frame of the invoker and returns control to the invoker.

Runtime Exceptions

If the current method is a **synchronized** method and the current thread is not the owner of the monitor acquired or reentered on invocation of the method, *areturn* throws an `IllegalMonitorStateException`. This can happen, for example, if a **synchronized** method contains a *monitorexit* instruction, but no *monitorenter* instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in [§8.13](#) and if the first of those rules is violated during invocation of the current method, then *areturn* throws an `IllegalMonitorStateException`.

arraylength

Operation

Get length of array

Format

<i>arraylength</i>

Forms

arraylength = 190 (0xbe)

Operand Stack

..., *arrayref* $\Rightarrow$..., *length*

Description

The *arrayref* must be of type **reference** and must refer to an array. It is popped from the operand stack. The *length* of the array it references is determined. That *length* is pushed onto the operand stack as an **int**.

Runtime Exception

If the *arrayref* is `null`, the *arraylength* instruction throws a `NullPointerException`.

astore

Operation

Store reference into local variable

Format



Forms

astore = 58 (0x3a)

Operand Stack

..., *objectref* $\Rightarrow$...

Description

The *index* is an unsigned byte that must be an index into the local variable array of the current frame ([§3.6](#)). The *objectref* on the top of the operand stack must be of type `returnAddress` or of type `reference`. It is popped from the operand stack, and the value of the local variable at *index* is set to *objectref*.

Notes

The *astore* instruction is used with an *objectref* of type `returnAddress` when implementing the `finally` clauses of the Java programming language (see [Section 7.13, "Compiling finally"](#)). The *aload* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction is intentional.

The *astore* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

astore_<n>

Operation

Store reference into local variable

Format

<code>astore_<n></code>

Forms

`astore_0` = 75 (0x4b) `astore_1` = 76 (0x4c) `astore_2` = 77 (0x4d) `astore_3` = 78 (0x4e)

Operand Stack

..., *objectref* $\Rightarrow$...

Description

The <n> must be an index into the local variable array of the current frame ([§3.6](#)). The *objectref* on the top of the operand stack must be of type `returnAddress` or of type `reference`. It is popped from the operand stack, and the value of the local variable at <n> is set to *objectref*.

Notes

An `astore_<n>` instruction is used with an *objectref* of type `returnAddress` when implementing the `finally` clauses of the Java programming language (see [Section 7.13, "Compiling finally"](#)). An `aload_<n>` instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the corresponding `astore_<n>` instruction is intentional.

Each of the `astore_<n>` instructions is the same as `astore` with an *index* of <n>, except that the operand <n> is implicit.

athrow

Operation

Throw exception or error

Format

<i>athrow</i>

Forms

athrow = 191 (0xbf)

Operand Stack

..., *objectref* $\Rightarrow$ *objectref*

Description

The *objectref* must be of type **reference** and must refer to an object that is an instance of class **Throwable** or of a subclass of **Throwable**. It is popped from the operand stack. The *objectref* is then thrown by searching the current method ([§3.6](#)) for the first exception handler that matches the class of *objectref*, as given by the algorithm in [§3.10](#).

If an exception handler that matches *objectref* is found, it contains the location of the code intended to handle this exception. The **pc** register is reset to that location, the operand stack of the current frame is cleared, *objectref* is pushed back onto the operand stack, and execution continues.

If no matching exception handler is found in the current frame, that frame is popped. If the current frame represents an invocation of a **synchronized** method, the monitor acquired or reentered on invocation of the method is released or exited (respectively) as if by execution of a *monitorexit* instruction. Finally, the frame of its invoker is reinstated, if such a frame exists, and the *objectref* is rethrown. If no such frame exists, the current thread exits.

Runtime Exceptions

If *objectref* is **null**, *athrow* throws a **NullPointerException** instead of *objectref*.

Otherwise, if the method of the current frame is a **synchronized** method and the current thread is not the owner of the monitor acquired or reentered on invocation of the method, *athrow* throws an **IllegalMonitorStateException** instead of the

object previously being thrown. This can happen, for example, if an abruptly completing `synchronized` method contains a *monitorexit* instruction, but no *monitorenter* instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in [§8.13](#) and if the first of those rules is violated during invocation of the current method, then *athrow* throws an `IllegalMonitorStateException` instead of the object previously being thrown.

Notes

The operand stack diagram for the *athrow* instruction may be misleading: If a handler for this exception is matched in the current method, the *athrow* instruction discards all the values on the operand stack, then pushes the thrown object onto the operand stack. However, if no handler is matched in the current method and the exception is thrown farther up the method invocation chain, then the operand stack of the method (if any) that handles the exception is cleared and *objectref* is pushed onto that empty operand stack. All intervening frames from the method that threw the exception up to, but not including, the method that handles the exception are discarded.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The JavaTM Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections to jvm@java.sun.com

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

d2f

Operation

Convert `double` to `float`

Format

<i>d2f</i>

Forms

d2f = 144 (0x90)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* on the top of the operand stack must be of type `double`. It is popped from the operand stack and undergoes value set conversion (§3.8.3) resulting in *value'*. Then *value'* is converted to a `float` *result* using IEEE 754 round to nearest mode. The *result* is pushed onto the operand stack.

Where an *d2f* instruction is FP-strict (§3.8.2), the result of the conversion is always rounded to the nearest representable value in the float value set (§3.3.2).

Where an *d2f* instruction is not FP-strict, the result of the conversion may be taken from the float-extended-exponent value set (§3.3.2); it is not necessarily rounded to the nearest representable value in the float value set.

A finite *value'* too small to be represented as a `float` is converted to a zero of the same sign; a finite *value'* too large to be represented as a `float` is converted to an infinity of the same sign. A `double` NaN is converted to a `float` NaN.

Notes

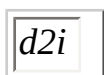
The *d2f* instruction performs a narrowing primitive conversion ([§2.6.3](#)). It may lose information about the overall magnitude of *value'* and may also lose precision.

d2i

Operation

Convert `double` to `int`

Format



Forms

d2i = 142 (0x8e)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* on the top of the operand stack must be of type `double`. It is popped from the operand stack and undergoes value set conversion ([§3.8.3](#)) resulting in *value'*. Then *value'* is converted to an `int`. The *result* is pushed onto the operand stack:

- If the *value'* is NaN, the *result* of the conversion is an `int` 0.
- Otherwise, if the *value'* is not an infinity, it is rounded to an integer value *V*, rounding towards zero using IEEE 754 round towards zero mode. If this integer value *V* can be represented as an `int`, then the *result* is the `int` value *V*.
- Otherwise, either the *value'* must be too small (a negative value of large magnitude or negative infinity), and the *result* is the smallest representable value of type `int`, or the *value'* must be too large (a positive value of large magnitude or positive infinity), and the *result* is the largest representable value of type `int`.

Notes

The *d2i* instruction performs a narrowing primitive conversion ([§2.6.3](#)). It may lose information about the overall magnitude of *value'* and may also lose precision.

d2l

Operation

Convert `double` to `long`

Format

<i>d2l</i>

Forms

d2l = 143 (0x8f)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* on the top of the operand stack must be of type `double`. It is popped from the operand stack and undergoes value set conversion ([§3.8.3](#)) resulting in *value'*. Then *value'* is converted to a `long`. The *result* is pushed onto the operand stack:

- If the *value'* is NaN, the *result* of the conversion is a `long` 0.
- Otherwise, if the *value'* is not an infinity, it is rounded to an integer value *V*, rounding towards zero using IEEE 754 round towards zero mode. If this integer value *V* can be represented as a `long`, then the *result* is the `long` value *V*.
- Otherwise, either the *value'* must be too small (a negative value of large magnitude or negative infinity), and the *result* is the smallest representable value of type `long`, or the *value'* must be too large (a positive value of large magnitude or positive infinity), and the *result* is the largest representable value of type `long`.

Notes

The *d2l* instruction performs a narrowing primitive conversion ([§2.6.3](#)). It may lose information about the overall magnitude of *value'* and may also lose precision.

dadd

Operation

Add double

Format

<i>dadd</i>

Forms

dadd = 99 (0x63)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type double. The values are popped from the operand stack and undergo value set conversion ([§3.8.3](#)), resulting in *value1'* and *value2'*. The double *result* is *value1' + value2'*. The *result* is pushed onto the operand stack.

The result of a *dadd* instruction is governed by the rules of IEEE arithmetic:

- If either *value1'* or *value2'* is NaN, the result is NaN.
- The sum of two infinities of opposite sign is NaN.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and any finite value is equal to the infinity.

- The sum of two zeroes of opposite sign is positive zero.
- The sum of two zeroes of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero value.
- The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.
- In the remaining cases, where neither operand is an infinity, a zero, or NaN and the values have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a `double`, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a `double`, we say the operation underflows; the result is then a zero of appropriate sign.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of a *dadd* instruction never throws a runtime exception.

dload

Operation

Load `double` from array

Format

<i>dload</i>

Forms

dload = 49 (0x31)

Operand Stack

..., *arrayref*, *index* $\Rightarrow$..., *value*

Description

The *arrayref* must be of type `reference` and must refer to an array whose

components are of type `double`. The *index* must be of type `int`. Both *arrayref* and *index* are popped from the operand stack. The `double` *value* in the component of the array at *index* is retrieved and pushed onto the operand stack.

Runtime Exceptions

If *arrayref* is `null`, *daload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *daload* instruction throws an `ArrayIndexOutOfBoundsException`.

dastore

Operation

Store into `double` array

Format

<i>dastore</i>

Forms

dastore = 82 (0x52)

Operand Stack

..., *arrayref*, *index*, *value* $\Rightarrow$...

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `double`. The *index* must be of type `int`, and *value* must be of type `double`. The *arrayref*, *index*, and *value* are popped from the operand stack. The `double` *value* undergoes value set conversion ([§3.8.3](#)), resulting in *value'*, which is stored as the component of the array indexed by *index*.

Runtime Exceptions

If *arrayref* is `null`, *dastore* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *dastore* instruction throws an `ArrayIndexOutOfBoundsException`.

dcmp<op>

Operation

Compare double

Format

<i>dcmp</i> < <i>op</i> >

Forms

dcmpg = 152 (0x98) *dcmpl* = 151 (0x97)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `double`. The values are popped from the operand stack and undergo value set conversion ([§3.8.3](#)), resulting in *value1'* and *value2'*. A floating-point comparison is performed:

- If *value1'* is greater than *value2'*, the `int` value *1* is pushed onto the operand stack.
- Otherwise, if *value1'* is equal to *value2'*, the `int` value *0* is pushed onto the operand stack.
- Otherwise, if *value1'* is less than *value2'*, the `int` value *-1* is pushed onto the operand stack.
- Otherwise, at least one of *value1'* or *value2'* is NaN. The *dcmpg* instruction pushes the `int` value *1* onto the operand stack and the *dcmpl* instruction pushes the `int` value *-1* onto the operand stack.

Floating-point comparison is performed in accordance with IEEE 754. All values other than NaN are ordered, with negative infinity less than all finite values and positive infinity greater than all finite values. Positive zero and negative zero are considered

equal.

Notes

The *dcmpg* and *dcmpl* instructions differ only in their treatment of a comparison involving NaN. NaN is unordered, so any **double** comparison fails if either or both of its operands are NaN. With both *dcmpg* and *dcmpl* available, any **double** comparison may be compiled to push the same *result* onto the operand stack whether the comparison fails on non-NaN values or fails because it encountered a NaN. For more information, see [Section 7.5, "More Control Examples."](#)

dconst_<d>

Operation

Push **double**

Format

<i>dconst_<d></i>

Forms

dconst_0 = 14 (0xe) *dconst_1* = 15 (0xf)

Operand Stack

... $\Rightarrow$..., <*d*>

Description

Push the **double** constant <*d*> (*0.0* or *1.0*) onto the operand stack.

ddiv

Operation

Divide double

Format

<i>ddiv</i>

Forms

ddiv = 111 (0x6f)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type double. The values are popped from the operand stack and undergo value set conversion ([§3.8.3](#)), resulting in *value1'* and *value2'*. The double *result* is *value1' / value2'*. The *result* is pushed onto the operand stack.

The result of a *ddiv* instruction is governed by the rules of IEEE arithmetic:

- If either *value1'* or *value2'* is NaN, the result is NaN.
- If neither *value1'* nor *value2'* is NaN, the sign of the result is positive if both values have the same sign, negative if the values have different signs.
- Division of an infinity by an infinity results in NaN.
- Division of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.
- Division of a finite value by an infinity results in a signed zero, with the sign-producing rule just given.
- Division of a zero by a zero results in NaN; division of zero by any other finite value results in a signed zero, with the sign-producing rule just given.
- Division of a nonzero finite value by a zero results in a signed infinity, with the sign-producing rule just given.

- In the remaining cases, where neither operand is an infinity, a zero, or NaN, the quotient is computed and rounded to the nearest `double` using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a `double`, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a `double`, we say the operation underflows; the result is then a zero of appropriate sign.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, division by zero, or loss of precision may occur, execution of a *ddiv* instruction never throws a runtime exception.

dload

Operation

Load `double` from local variable

Format



Forms

dload = 24 (0x18)

Operand Stack

... $\Rightarrow$..., *value*

Description

The *index* is an unsigned byte. Both *index* and *index* + 1 must be indices into the local variable array of the current frame ([§3.6](#)). The local variable at *index* must contain a `double`. The *value* of the local variable at *index* is pushed onto the operand stack.

Notes

The *dload* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

dload_<n>

Operation

Load **double** from local variable

Format

<i>dload_<n></i>

Forms

dload_0 = 38 (0x26) *dload_1* = 39 (0x27) *dload_2* = 40 (0x28) *dload_3* = 41 (0x29)

Operand Stack

... $\Rightarrow$..., *value*

Description

Both <*n*> and <*n*> + 1 must be indices into the local variable array of the current frame ([§3.6](#)). The local variable at <*n*> must contain a **double**. The *value* of the local variable at <*n*> is pushed onto the operand stack.

Notes

Each of the *dload_<n>* instructions is the same as *dload* with an *index* of <*n*>, except that the operand <*n*> is implicit.

dmul

Operation

Multiply **double**

Format



Forms

dmul = 107 (0x6b)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `double`. The values are popped from the operand stack and undergo value set conversion ([§3.8.3](#)), resulting in *value1'* and *value2'*. The `double` *result* is *value1' * value2'*. The *result* is pushed onto the operand stack.

The result of a *dmul* instruction is governed by the rules of IEEE arithmetic:

- If either *value1'* or *value2'* is NaN, the result is NaN.
- If neither *value1'* nor *value2'* is NaN, the sign of the result is positive if both values have the same sign and negative if the values have different signs.
- Multiplication of an infinity by a zero results in NaN.
- Multiplication of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.
- In the remaining cases, where neither an infinity nor NaN is involved, the product is computed and rounded to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a `double`, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a `double`, we say the operation underflows; the result is then a zero of appropriate sign.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of a *dmul* instruction never throws a runtime exception.

dneg

Operation

Negate double

Format

dneg

Forms

dneg = 119 (0x77)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* must be of type **double**. It is popped from the operand stack and undergoes value set conversion ([§3.8.3](#)), resulting in *value'*. The **double** *result* is the arithmetic negation of *value'*. The *result* is pushed onto the operand stack.

For **double** values, negation is not the same as subtraction from zero. If *x* is $+0.0$, then $0.0 - x$ equals $+0.0$, but $-x$ equals -0.0 . Unary minus merely inverts the sign of a **double**.

Special cases of interest:

- If the operand is NaN, the result is NaN (recall that NaN has no sign).
- If the operand is an infinity, the result is the infinity of opposite sign.
- If the operand is a zero, the result is the zero of opposite sign.

drem

Operation

Remainder double

Format

drem

Forms

drem = 115 (0x73)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type **double**. The values are popped from the operand stack and undergo value set conversion ([§3.8.3](#)), resulting in *value1'* and *value2'*. The *result* is calculated and pushed onto the operand stack as a **double**.

The result of a *drem* instruction is not the same as that of the so-called remainder operation defined by IEEE 754. The IEEE 754 "remainder" operation computes the remainder from a rounding division, not a truncating division, and so its behavior is *not* analogous to that of the usual integer remainder operator. Instead, the Java virtual machine defines *drem* to behave in a manner analogous to that of the Java virtual machine integer remainder instructions (*irem* and *lrem*); this may be compared with the C library function `fmod`.

The result of a *drem* instruction is governed by these rules:

- If either *value1'* or *value2'* is NaN, the result is NaN.
- If neither *value1'* nor *value2'* is NaN, the sign of the result equals the sign of the dividend.
- If the dividend is an infinity or the divisor is a zero or both, the result is NaN.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
- If the dividend is a zero and the divisor is finite, the result equals the dividend.
- In the remaining cases, where neither operand is an infinity, a zero, or NaN, the floating-point remainder *result* from a dividend *value1'* and a divisor *value2'* is defined by the mathematical relation $result = value1' - (value2' * q)$, where *q* is an integer that is negative only if *value1' / value2'* is negative, and positive only if *value1' / value2'* is positive, and whose magnitude is

as large as possible without exceeding the magnitude of the true mathematical quotient of *value1'* and *value2'*.

Despite the fact that division by zero may occur, evaluation of a *drem* instruction never throws a runtime exception. Overflow, underflow, or loss of precision cannot occur.

Notes

The IEEE 754 remainder operation may be computed by the library routine `Math.IEEEremainder`.

dreturn

Operation

Return `double` from method

Format

<i>dreturn</i>

Forms

dreturn = 175 (0xaf)

Operand Stack

..., *value* $\Rightarrow$ [empty]

Description

The current method must have return type `double`. The *value* must be of type `double`. If the current method is a **synchronized** method, the monitor acquired or reentered on invocation of the method is released or exited (respectively) as if by execution of a *monitorexit* instruction. If no exception is thrown, *value* is popped from the operand stack of the current frame ([§3.6](#)) and undergoes value set conversion ([§3.8.3](#)), resulting in *value'*. The *value'* is pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The interpreter then returns control to the invoker of the method, reinstating the frame

of the invoker.

Runtime Exceptions

If the current method is a **synchronized** method and the current thread is not the owner of the monitor acquired or reentered on invocation of the method, *dreturn* throws an `IllegalMonitorStateException`. This can happen, for example, if a **synchronized** method contains a *monitorexit* instruction, but no *monitorenter* instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in [§8.13](#) and if the first of those rules is violated during invocation of the current method, then *dreturn* throws an `IllegalMonitorStateException`.

dstore

Operation

Store `double` into local variable

Format



Forms

dstore = 57 (0x39)

Operand Stack

..., *value* $\Rightarrow$...

Description

The *index* is an unsigned byte. Both *index* and *index* + 1 must be indices into the local variable array of the current frame ([§3.6](#)). The *value* on the top of the operand stack must be of type `double`. It is popped from the operand stack and undergoes value set conversion ([§3.8.3](#)), resulting in *value'*. The local variables at *index* and *index* + 1 are

set to *value'*.

Notes

The *dstore* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

dstore_<n>

Operation

Store `double` into local variable

Format

<i>dstore_<n></i>

Forms

dstore_0 = 71 (0x47) *dstore_1* = 72 (0x48) *dstore_2* = 73 (0x49) *dstore_3* = 74 (0x4a)

Operand Stack

..., *value* $\Rightarrow$...

Description

Both *<n>* and *<n> + 1* must be indices into the local variable array of the current frame ([§3.6](#)). The *value* on the top of the operand stack must be of type `double`. It is popped from the operand stack and undergoes value set conversion ([§3.8.3](#)), resulting in *value'*. The local variables at *<n>* and *<n> + 1* are set to *value'*.

Notes

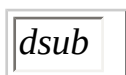
Each of the *dstore_<n>* instructions is the same as *dstore* with an *index* of *<n>*, except that the operand *<n>* is implicit.

dsub

Operation

Subtract `double`

Format



Forms

dsub = 103 (0x67)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `double`. The values are popped from the operand stack and undergo value set conversion ([§3.8.3](#)), resulting in *value1'* and *value2'*. The `double` *result* is *value1'* - *value2'*. The *result* is pushed onto the operand stack.

For `double` subtraction, it is always the case that *a-b* produces the same result as *a+(-b)*. However, for the *dsub* instruction, subtraction from zero is not the same as negation, because if *x* is `+0.0`, then `0.0-x` equals `+0.0`, but `-x` equals `-0.0`.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of a *dsub* instruction never throws a runtime exception.

dup

Operation

Duplicate the top operand stack value

Format

dup

Forms

dup = 89 (0x59)

Operand Stack

$..., value \Rightarrow ..., value, value$

Description

Duplicate the top value on the operand stack and push the duplicated value onto the operand stack.

The *dup* instruction must not be used unless *value* is a value of a category 1 computational type ([§3.11.1](#)).

dup_x1

Operation

Duplicate the top operand stack value and insert two values down

Format

dup_x1

Forms

dup_x1 = 90 (0x5a)

Operand Stack

$..., value2, value1 \Rightarrow ..., value1, value2, value1$

Description

Duplicate the top value on the operand stack and insert the duplicated value two values down in the operand stack.

The *dup_x1* instruction must not be used unless both *value1* and *value2* are values of a category 1 computational type ([§3.11.1](#)).

dup_x2

Operation

Duplicate the top operand stack value and insert two or three values down

Format

<i>dup_x2</i>

Forms

dup_x2 = 91 (0x5b)

Operand Stack

Form 1:

$..., value3, value2, value1 \Rightarrow ..., value1, value3, value2, value1$

where *value1*, *value2*, and *value3* are all values of a category 1 computational type ([§3.11.1](#)).

Form 2:

$..., value2, value1 \Rightarrow ..., value1, value2, value1$

where *value1* is a value of a category 1 computational type and *value2* is a value of a category 2 computational type ([§3.11.1](#)).

Description

Duplicate the top value on the operand stack and insert the duplicated value two or

three values down in the operand stack.

dup2

Operation

Duplicate the top one or two operand stack values

Format

<i>dup2</i>

Forms

dup2 = 92 (0x5c)

Operand Stack

Form 1:

$..., value2, value1 \Rightarrow ..., value2, value1, value2, value1$

where both *value1* and *value2* are values of a category 1 computational type ([§3.11.1](#)).

Form 2:

$..., value \Rightarrow ..., value, value$

where *value* is a value of a category 2 computational type ([§3.11.1](#)).

Description

Duplicate the top one or two values on the operand stack and push the duplicated value or values back onto the operand stack in the original order.

dup2_x1

Operation

Duplicate the top one or two operand stack values and insert two or three values down

Format

<i>dup2_x1</i>

Forms

dup2_x1 = 93 (0x5d)

Operand Stack

Form 1:

$..., value3, value2, value1 \Rightarrow ..., value2, value1, value3, value2, value1$

where *value1*, *value2*, and *value3* are all values of a category 1 computational type [\(§3.11.1\)](#).

Form 2:

$..., value2, value1 \Rightarrow ..., value1, value2, value1$

where *value1* is a value of a category 2 computational type and *value2* is a value of a category 1 computational type [\(§3.11.1\)](#).

Description

Duplicate the top one or two values on the operand stack and insert the duplicated values, in the original order, one value beneath the original value or values in the operand stack.

dup2_x2

Operation

Duplicate the top one or two operand stack values and insert two, three, or four values down

Format

dup2_x2

Forms

dup2_x2 = 94 (0x5e)

Operand Stack

Form 1:

$..., value4, value3, value2, value1 \Rightarrow ..., value2, value1, value4, value3, value2, value1$

where *value1*, *value2*, *value3*, and *value4* are all values of a category 1 computational type ([§3.11.1](#)).

Form 2:

$..., value3, value2, value1 \Rightarrow ..., value1, value3, value2, value1$

where *value1* is a value of a category 2 computational type and *value2* and *value3* are both values of a category 1 computational type ([§3.11.1](#)).

Form 3:

$..., value3, value2, value1 \Rightarrow ..., value2, value1, value3, value2, value1$

where *value1* and *value2* are both values of a category 1 computational type and *value3* is a value of a category 2 computational type ([§3.11.1](#)).

Form 4:

$..., value2, value1 \Rightarrow ..., value1, value2, value1$

where *value1* and *value2* are both values of a category 2 computational type ([§3.11.1](#)).

Description

Duplicate the top one or two values on the operand stack and insert the duplicated

values, in the original order, into the operand stack.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The JavaTM Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections to jvm@java.sun.com

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

f2d

Operation

Convert `float` to `double`

Format

<i>f2d</i>

Forms

f2d = 141 (0x8d)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* on the top of the operand stack must be of type `float`. It is popped from the operand stack and undergoes value set conversion ([§3.8.3](#)), resulting in *value'*. Then *value'* is converted to a `double` *result*. This *result* is pushed onto the operand stack.

Notes

Where an *f2d* instruction is FP-strict ([§3.8.2](#)) it performs a widening primitive conversion ([§2.6.2](#)). Because all values of the float value set ([§3.3.2](#)) are exactly representable by values of the double value set ([§3.3.2](#)), such a conversion is exact.

Where an *f2d* instruction is not FP-strict, the result of the conversion may be taken from the double-extended-exponent value set; it is not necessarily rounded to the nearest representable value in the double value set. However, if the operand *value* is taken from the float-extended-exponent value set and the target result is constrained to the double value set, rounding of *value* may be required.

f2i

Operation

Convert `float` to `int`

Format



Forms

`f2i` = 139 (0x8b)

Operand Stack

$..., value \Rightarrow ..., result$

Description

The *value* on the top of the operand stack must be of type `float`. It is popped from the operand stack and undergoes value set conversion ([§3.8.3](#)), resulting in *value'*. Then *value'* is converted to an `int` *result*. This *result* is pushed onto the operand stack:

- If the *value'* is NaN, the *result* of the conversion is an `int` 0.
- Otherwise, if the *value'* is not an infinity, it is rounded to an integer value *V*, rounding towards zero using IEEE 754 round towards zero mode. If this integer value *V* can be represented as an `int`, then the *result* is the `int` value *V*.
- Otherwise, either the *value'* must be too small (a negative value of large magnitude or negative infinity), and the *result* is the smallest representable value of type `int`, or the *value'* must be too large (a positive value of large magnitude or positive infinity), and the *result* is the largest representable value of type `int`.

Notes

The `f2i` instruction performs a narrowing primitive conversion ([§2.6.3](#)). It may lose information about the overall magnitude of *value'* and may also lose precision.

f2l

Operation

Convert `float` to `long`

Format



Forms

`f2l` = 140 (0x8c)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* on the top of the operand stack must be of type `float`. It is popped from the operand stack and undergoes value set conversion ([§3.8.3](#)), resulting in *value'*. Then *value'* is converted to a `long` *result*. This *result* is pushed onto the operand stack:

- If the *value'* is NaN, the *result* of the conversion is a `long` 0.
- Otherwise, if the *value'* is not an infinity, it is rounded to an integer value *V*, rounding towards zero using IEEE 754 round towards zero mode. If this integer value *V* can be represented as a `long`, then the *result* is the `long` value *V*.
- Otherwise, either the *value'* must be too small (a negative value of large magnitude or negative infinity), and the *result* is the smallest representable value of type `long`, or the *value'* must be too large (a positive value of large magnitude or positive infinity), and the *result* is the largest representable value of type `long`.

Notes

The `f2l` instruction performs a narrowing primitive conversion ([§2.6.3](#)). It may lose information about the overall magnitude of *value'* and may also lose precision.

fadd

Operation

Add `f l o a t`

Format



Forms

fadd = 98 (0x62)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `f l o a t`. The values are popped from the operand stack and undergo value set conversion ([§3.8.3](#)), resulting in *value1'* and *value2'*. The `f l o a t` *result* is *value1'* + *value2'*. The *result* is pushed onto the operand stack.

The result of an *fadd* instruction is governed by the rules of IEEE arithmetic:

- If either *value1'* or *value2'* is NaN, the result is NaN.
- The sum of two infinities of opposite sign is NaN.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and any finite value is equal to the infinity.
- The sum of two zeroes of opposite sign is positive zero.
- The sum of two zeroes of the same sign is the zero of that sign.

- The sum of a zero and a nonzero finite value is equal to the nonzero value.
- The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.
- In the remaining cases, where neither operand is an infinity, a zero, or NaN and the values have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a `float`, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a `float`, we say the operation underflows; the result is then a zero of appropriate sign.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of an *fadd* instruction never throws a runtime exception.

faload

Operation

Load `float` from array

Format

<i>faload</i>

Forms

faload = 48 (0x30)

Operand Stack

..., *arrayref*, *index* $\Rightarrow$..., *value*

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `float`. The *index* must be of type `int`. Both *arrayref* and *index* are popped from the operand stack. The `float` *value* in the component of the array at *index* is retrieved and pushed onto the operand stack.

Runtime Exceptions

If *arrayref* is `null`, *faload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *faload* instruction throws an `ArrayIndexOutOfBoundsException`.

fastore

Operation

Store into `float` array

Format

<i>fastore</i>

Forms

fastore = 81 (0x51)

Operand Stack

..., *arrayref*, *index*, *value* $\Rightarrow$...

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `float`. The *index* must be of type `int`, and the *value* must be of type `float`. The *arrayref*, *index*, and *value* are popped from the operand stack. The `float` *value* undergoes value set conversion ([§3.8.3](#)), resulting in *value'*, and *value'* is stored as the component of the array indexed by *index*.

Runtime Exceptions

If *arrayref* is `null`, *fastore* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *fastore* instruction throws an `ArrayIndexOutOfBoundsException`.

fcmp<op>

Operation

Compare `float`

Format

<i>fcmp<op></i>

Forms

fcmpg = 150 (0x96) *fcmpl* = 149 (0x95)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `float`. The values are popped from the operand stack and undergo value set conversion ([§3.8.3](#)), resulting in *value1'* and *value2'*. A floating-point comparison is performed:

- If *value1'* is greater than *value2'*, the `int` value *1* is pushed onto the operand stack.
- Otherwise, if *value1'* is equal to *value2'*, the `int` value *0* is pushed onto the operand stack.
- Otherwise, if *value1'* is less than *value2'*, the `int` value *-1* is pushed onto the operand stack.
- Otherwise, at least one of *value1'* or *value2'* is NaN. The *fcmpg* instruction pushes the `int` value *1* onto the operand stack and the *fcmpl* instruction pushes the `int` value *-1* onto the operand stack.

Floating-point comparison is performed in accordance with IEEE 754. All values other than NaN are ordered, with negative infinity less than all finite values and positive infinity greater than all finite values. Positive zero and negative zero are considered equal.

Notes

The *fcmpg* and *fcmpl* instructions differ only in their treatment of a comparison involving NaN. NaN is unordered, so any `float` comparison fails if either or both of its operands are NaN. With both *fcmpg* and *fcmpl* available, any `float` comparison may be compiled to push the same *result* onto the operand stack whether the comparison fails on non-NaN values or fails because it encountered a NaN. For more information, see [Section 7.5, "More Control Examples."](#)

fconst_<f>

Operation

Push `float`

Format

<i>fconst_<f></i>

Forms

fconst_0 = 11 (0xb) *fconst_1* = 12 (0xc) *fconst_2* = 13 (0xd)

Operand Stack

... $\Rightarrow$..., <f>

Description

Push the `float` constant <f> (0.0, 1.0, or 2.0) onto the operand stack.

fdiv

Operation

Divide `float`

Format



Forms

fdiv = 110 (0x6e)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `float`. The values are popped from the operand stack and undergo value set conversion ([§3.8.3](#)), resulting in *value1'* and *value2'*. The `float` *result* is *value1' / value2'*. The *result* is pushed onto the operand stack.

The result of an *fdiv* instruction is governed by the rules of IEEE arithmetic:

- If either *value1'* or *value2'* is NaN, the result is NaN.
- If neither *value1'* nor *value2'* is NaN, the sign of the result is positive if both values have the same sign, negative if the values have different signs.
- Division of an infinity by an infinity results in NaN.
- Division of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.
- Division of a finite value by an infinity results in a signed zero, with the sign-producing rule just given.
- Division of a zero by a zero results in NaN; division of zero by any other finite value results in a signed zero, with the sign-producing rule just given.
- Division of a nonzero finite value by a zero results in a signed infinity, with the sign-producing rule just given.
- In the remaining cases, where neither operand is an infinity, a zero, or NaN, the quotient is computed and rounded to the nearest `float` using IEEE 754 round to nearest mode. If the

magnitude is too large to represent as a `float`, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a `float`, we say the operation underflows; the result is then a zero of appropriate sign.

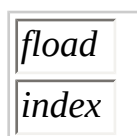
The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, division by zero, or loss of precision may occur, execution of an *fdiv* instruction never throws a runtime exception.

float

Operation

Load `float` from local variable

Format



Forms

fload = 23 (0x17)

Operand Stack

... $\Rightarrow$..., *value*

Description

The *index* is an unsigned byte that must be an index into the local variable array of the current frame ([§3.6](#)). The local variable at *index* must contain a `float`. The *value* of the local variable at *index* is pushed onto the operand stack.

Notes

The *fload* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

fload_<n>

Operation

Load `fload` from local variable

Format

<i>fload_<n></i>

Forms

fload_0 = 34 (0x22) *fload_1* = 35 (0x23) *fload_2* = 36 (0x24) *fload_3* = 37 (0x25)

Operand Stack

... $\Rightarrow$..., *value*

Description

The *<n>* must be an index into the local variable array of the current frame ([§3.6](#)). The local variable at *<n>* must contain a `fload`. The *value* of the local variable at *<n>* is pushed onto the operand stack.

Notes

Each of the *fload_<n>* instructions is the same as *fload* with an *index* of *<n>*, except that the operand *<n>* is implicit.

fmul

Operation

Multiply `fload`

Format

<i>fmul</i>

Forms

fmul = 106 (0x6a)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `float`. The values are popped from the operand stack and undergo value set conversion ([§3.8.3](#)), resulting in *value1'* and *value2'*. The `float` *result* is *value1' * value2'*. The *result* is pushed onto the operand stack.

The result of an *fmul* instruction is governed by the rules of IEEE arithmetic:

- If either *value1'* or *value2'* is NaN, the result is NaN.
- If neither *value1'* nor *value2'* is NaN, the sign of the result is positive if both values have the same sign, and negative if the values have different signs.
- Multiplication of an infinity by a zero results in NaN.
- Multiplication of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.
- In the remaining cases, where neither an infinity nor NaN is involved, the product is computed and rounded to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a `float`, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a `float`, we say the operation underflows; the result is then a zero of appropriate sign.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of an *fmul* instruction never throws a runtime exception.

fneg

Operation

Negate float

Format



Forms

fneg = 118 (0x76)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* must be of type `float`. It is popped from the operand stack and undergoes value set conversion ([§3.8.3](#)), resulting in *value'*. The `float` *result* is the arithmetic negation of *value'*. This *result* is pushed onto the operand stack.

For `float` values, negation is not the same as subtraction from zero. If *x* is `+0.0`, then `0.0-x` equals `+0.0`, but `-x` equals `-0.0`. Unary minus merely inverts the sign of a `float`.

Special cases of interest:

- If the operand is NaN, the result is NaN (recall that NaN has no sign).
- If the operand is an infinity, the result is the infinity of opposite sign.
- If the operand is a zero, the result is the zero of opposite sign.

frem

Operation

Remainder float

Format



Forms

frem = 114 (0x72)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `float`. The values are popped from the operand stack and undergo value set conversion ([§3.8.3](#)), resulting in *value1'* and *value2'*. The *result* is calculated and pushed onto the operand stack as a `float`.

The *result* of an *frem* instruction is not the same as that of the so-called remainder operation defined by IEEE 754. The IEEE 754 "remainder" operation computes the remainder from a rounding division, not a truncating division, and so its behavior is *not* analogous to that of the usual integer remainder operator. Instead, the Java virtual machine defines *frem* to behave in a manner analogous to that of the Java virtual machine integer remainder instructions (*irem* and *lrem*); this may be compared with the C library function `fmod`.

The result of an *frem* instruction is governed by these rules:

- If either *value1'* or *value2'* is NaN, the result is NaN.
- If neither *value1'* nor *value2'* is NaN, the sign of the result equals the sign of the dividend.
- If the dividend is an infinity or the divisor is a zero or both, the result is NaN.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
- If the dividend is a zero and the divisor is finite, the result equals the dividend.
- In the remaining cases, where neither operand is an infinity, a zero, or NaN, the floating-point remainder *result* from a dividend *value1'* and a divisor *value2'* is defined by the mathematical relation $result = value1' - (value2' * q)$, where *q* is an integer that is negative only if *value1' / value2'* is negative and positive only if *value1' / value2'* is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of *value1'* and *value2'*.

Despite the fact that division by zero may occur, evaluation of an *frem* instruction never throws a runtime exception. Overflow, underflow, or loss of precision cannot occur.

Notes

The IEEE 754 remainder operation may be computed by the library routine `Math.IEEEremainder`.

freturn

Operation

Return `float` from method

Format

<i>freturn</i>

Forms

freturn = 174 (0xae)

Operand Stack

..., *value* $\Rightarrow$ [empty]

Description

The current method must have return type `float`. The *value* must be of type `float`. If the current method is a `synchronized` method, the monitor acquired or reentered on invocation of the method is released or exited (respectively) as if by execution of a *monitorexit* instruction. If no exception is thrown, *value* is popped from the operand stack of the current frame ([§3.6](#)) and undergoes value set conversion ([§3.8.3](#)), resulting in *value'*. The *value'* is pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

Runtime Exceptions

If the current method is a **synchronized** method and the current thread is not the owner of the monitor acquired or reentered on invocation of the method, *freturn* throws an `IllegalMonitorStateException`. This can happen, for example, if a **synchronized** method contains a *monitorexit* instruction, but no *monitorenter* instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in [§8.13](#) and if the first of those rules is violated during invocation of the current method, then *freturn* throws an `IllegalMonitorStateException`.

fstore

Operation

Store `float` into local variable

Format



Forms

fstore = 56 (0x38)

Operand Stack

..., *value* $\Rightarrow$...

Description

The *index* is an unsigned byte that must be an index into the local variable array of the current frame ([§3.6](#)). The *value* on the top of the operand stack must be of type `float`. It is popped from the operand stack and undergoes value set conversion ([§3.8.3](#)), resulting in *value'*. The value of the local variable at *index* is set to *value'*.

Notes

The *fstore* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

fstore_<n>

Operation

Store `f l o a t` into local variable

Format

<i>fstore_<n></i>

Forms

fstore_0 = 67 (0x43) *fstore_1* = 68 (0x44) *fstore_2* = 69 (0x45) *fstore_3* = 70 (0x46)

Operand Stack

..., *value* $\Rightarrow$...

Description

The *<n>* must be an index into the local variable array of the current frame ([§3.6](#)). The *value* on the top of the operand stack must be of type `f l o a t`. It is popped from the operand stack and undergoes value set conversion ([§3.8.3](#)), resulting in *value'*. The value of the local variable at *<n>* is set to *value'*.

Notes

Each of the *fstore_<n>* is the same as *fstore* with an *index* of *<n>*, except that the operand *<n>* is implicit.

fsub

Operation

Subtract `f l o a t`

Format

<i>fsub</i>

Forms

fsub = 102 (0x66)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `f l o a t`. The values are popped from the operand stack and undergo value set conversion ([§3.8.3](#)), resulting in *value1'* and *value2'*. The `f l o a t` *result* is *value1'* - *value2'*. The *result* is pushed onto the operand stack.

For `f l o a t` subtraction, it is always the case that *a*-*b* produces the same result as *a*+(-*b*). However, for the *fsub* instruction, subtraction from zero is not the same as negation, because if *x* is +0.0, then 0.0-*x* equals +0.0, but -*x* equals -0.0.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of an *fsub* instruction never throws a runtime exception.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The JavaTM Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections to jvm@java.sun.com

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

i2b

Operation

Convert `int` to `byte`

Format

<i>i2b</i>

Forms

i2b = 145 (0x91)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack, truncated to a `byte`, then sign-extended to an `int` *result*. That *result* is pushed onto the operand stack.

Notes

The *i2b* instruction performs a narrowing primitive conversion ([§2.6.3](#)). It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign as *value*.

i2c

Operation

Convert `int` to `char`

Format

<i>i2c</i>

Forms

i2c = 146 (0x92)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack, truncated to `char`, then zero-extended to an `int` *result*. That *result* is pushed onto the operand stack.

Notes

The *i2c* instruction performs a narrowing primitive conversion ([§2.6.3](#)). It may lose information about the overall magnitude of *value*. The *result* (which is always positive) may also not have the same sign as *value*.

i2d

Operation

Convert `int` to `double`

Format

<i>i2d</i>

Forms

$i2d = 135$ (0x87)

Operand Stack

$\dots, value \Rightarrow \dots, result$

Description

The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack and converted to a `double` *result*. The *result* is pushed onto the operand stack.

Notes

The *i2d* instruction performs a widening primitive conversion ([§2.6.2](#)). Because all values of type `int` are exactly representable by type `double`, the conversion is exact.

i2f

Operation

Convert `int` to `float`

Format



Forms

$i2f = 134$ (0x86)

Operand Stack

$\dots, value \Rightarrow \dots, result$

Description

The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack and converted to the `float` *result* using IEEE 754 round to nearest mode. The *result* is pushed onto the operand stack.

Notes

The *i2f* instruction performs a widening primitive conversion ([§2.6.2](#)), but may result in a loss of precision because values of type `float` have only 24 significand bits.

i2l

Operation

Convert `int` to `long`

Format



Forms

i2l = 133 (0x85)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack and sign-extended to a `long` *result*. That *result* is pushed onto the operand stack.

Notes

The *i2l* instruction performs a widening primitive conversion ([§2.6.2](#)). Because all values of type `int` are exactly representable by type `long`, the conversion is exact.

i2s

Operation

Convert `int` to `short`

Format

<i>i2s</i>

Forms

i2s = 147 (0x93)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack, truncated to a `short`, then sign-extended to an `int` *result*. That *result* is pushed onto the operand stack.

Notes

The *i2s* instruction performs a narrowing primitive conversion ([§2.6.3](#)). It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign as *value*.

iadd

Operation

Add `int`

Format

<i>iadd</i>

Forms

iadd = 96 (0x60)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *iadd* instruction never throws a runtime exception.

iaload

Operation

Load `int` from array

Format

<i>iaload</i>

Forms

iaload = 46 (0x2e)

Operand Stack

..., *arrayref*, *index* $\Rightarrow$..., *value*

Description

The *arrayref* must be of type **reference** and must refer to an array whose components are of type **int**. The *index* must be of type **int**. Both *arrayref* and *index* are popped from the operand stack. The **int** *value* in the component of the array at *index* is retrieved and pushed onto the operand stack.

Runtime Exceptions

If *arrayref* is **null**, *iaload* throws a **NullPointerException**.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *iaload* instruction throws an **ArrayIndexOutOfBoundsException**.

iand

Operation

Boolean AND **int**

Format

<i>iand</i>

Forms

iand = 126 (0x7e)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type **int**. They are popped from the operand stack. An **int** *result* is calculated by taking the bitwise AND (conjunction) of *value1* and *value2*. The *result* is pushed onto the operand stack.

iastore

Operation

Store into `int` array

Format

<i>iastore</i>

Forms

iastore = 79 (0x4f)

Operand Stack

..., *arrayref*, *index*, *value* $\Rightarrow$...

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `int`. Both *index* and *value* must be of type `int`. The *arrayref*, *index*, and *value* are popped from the operand stack. The `int` *value* is stored as the component of the array indexed by *index*.

Runtime Exceptions

If *arrayref* is `null`, *iastore* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *iastore* instruction throws an `ArrayIndexOutOfBoundsException`.

`iconst_<i>`

Operation

Push `int` constant

Format

<i>iconst_<i></i>

Forms

iconst_m1 = 2 (0x2) *iconst_0* = 3 (0x3) *iconst_1* = 4 (0x4) *iconst_2* = 5 (0x5) *iconst_3* = 6 (0x6) *iconst_4* = 7 (0x7) *iconst_5* = 8 (0x8)

Operand Stack

... $\Rightarrow$..., <*i*>

Description

Push the `int` constant <*i*> (-1, 0, 1, 2, 3, 4 or 5) onto the operand stack.

Notes

Each of this family of instructions is equivalent to *bipush* <*i*> for the respective value of <*i*>, except that the operand <*i*> is implicit.

idiv

Operation

Divide `int`

Format

<i>idiv</i>

Forms

idiv = 108 (0x6c)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is the value of the Java programming language expression *value1* / *value2*. The *result* is pushed onto the operand stack.

An `int` division rounds towards 0; that is, the quotient produced for `int` values in n/d is an `int` value q whose magnitude is as large as possible while satisfying $|d \cdot q| \leq |n|$. Moreover, q is positive when $|n| \geq |d|$ and n and d have the same sign, but q is negative when $|n| \geq |d|$ and n and d have opposite signs.

There is one special case that does not satisfy this rule: if the dividend is the negative integer of largest possible magnitude for the `int` type, and the divisor is `-1`, then overflow occurs, and the result is equal to the dividend. Despite the overflow, no exception is thrown in this case.

Runtime Exception

If the value of the divisor in an `int` division is 0, *idiv* throws an `ArithmeticException`.

if_acmp<cond>

Operation

Branch if reference comparison succeeds

Format

<i>if_acmp<cond></i>
<i>branchbyte1</i>
<i>branchbyte2</i>

Forms

if_acmpeq = 165 (0xa5) *if_acmpne* = 166 (0xa6)

Operand Stack

..., *value1*, *value2* $\Rightarrow$...

Description

Both *value1* and *value2* must be of type **reference**. They are both popped from the operand stack and compared. The results of the comparison are as follows:

- *eq* succeeds if and only if *value1* = *value2*
- *ne* succeeds if and only if *value1* $\neq$ *value2*

If the comparison succeeds, the unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is calculated to be (*branchbyte1* << 8) | *branchbyte2*. Execution then proceeds at that offset from the address of the opcode of this *if_acmp*<*cond*> instruction. The target address must be that of an opcode of an instruction within the method that contains this *if_acmp*<*cond*> instruction.

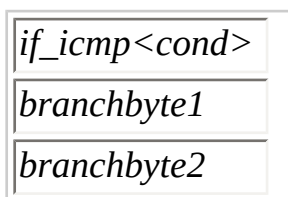
Otherwise, if the comparison fails, execution proceeds at the address of the instruction following this *if_acmp*<*cond*> instruction.

if_icmp<cond>

Operation

Branch if **int** comparison succeeds

Format



Forms

if_icmpeq = 159 (0x9f) *if_icmpne* = 160 (0xa0) *if_icmplt* = 161 (0xa1) *if_icmpge* = 162 (0xa2) *if_icmpgt* = 163 (0xa3) *if_icmple* = 164 (0xa4)

Operand Stack

..., *value1*, *value2* $\Rightarrow$...

Description

Both *value1* and *value2* must be of type `int`. They are both popped from the operand stack and compared. All comparisons are signed. The results of the comparison are as follows:

- *eq* succeeds if and only if *value1* = *value2*
- *ne* succeeds if and only if *value1* $\neq$ *value2*
- *lt* succeeds if and only if *value1* < *value2*
- *le* succeeds if and only if *value1* $\leq$ *value2*
- *gt* succeeds if and only if *value1* > *value2*
- *ge* succeeds if and only if *value1* $\geq$ *value2*

If the comparison succeeds, the unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is calculated to be (*branchbyte1* << 8) | *branchbyte2*. Execution then proceeds at that offset from the address of the opcode of this *if_icmp<cond>* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if_icmp<cond>* instruction.

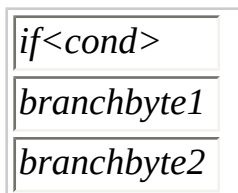
Otherwise, execution proceeds at the address of the instruction following this *if_icmp<cond>* instruction.

if<cond>

Operation

Branch if `int` comparison with zero succeeds

Format



Forms

ifeq = 153 (0x99) *ifne* = 154 (0x9a) *iflt* = 155 (0x9b) *ifge* = 156 (0x9c) *ifgt* = 157 (0x9d) *ifle* = 158 (0x9e)

Operand Stack

..., *value* $\Rightarrow$...

Description

The *value* must be of type `int`. It is popped from the operand stack and compared against zero. All comparisons are signed. The results of the comparisons are as follows:

- *eq* succeeds if and only if *value* = 0
- *ne* succeeds if and only if *value* $\neq$ 0
- *lt* succeeds if and only if *value* < 0
- *le* succeeds if and only if *value* $\leq$ 0
- *gt* succeeds if and only if *value* > 0
- *ge* succeeds if and only if *value* $\geq$ 0

If the comparison succeeds, the *unsigned branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is calculated to be (*branchbyte1* << 8) | *branchbyte2*. Execution then proceeds at that offset from the address of the opcode of this *if<cond>* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if<cond>* instruction.

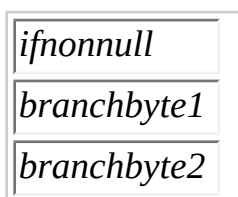
Otherwise, execution proceeds at the address of the instruction following this *if<cond>* instruction.

ifnonnull

Operation

Branch if reference not null

Format



Forms

ifnonnull = 199 (0xc7)

Operand Stack

..., *value* $\Rightarrow$...

Description

The *value* must be of type **reference**. It is popped from the operand stack. If *value* is not `null`, the *unsigned branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is calculated to be $(branchbyte1 \ll 8) \mid branchbyte2$. Execution then proceeds at that offset from the address of the opcode of this *ifnonnull* instruction. The target address must be that of an opcode of an instruction within the method that contains this *ifnonnull* instruction.

Otherwise, execution proceeds at the address of the instruction following this *ifnonnull* instruction.

ifnull

Operation

Branch if **reference** is `null`

Format

<i>ifnull</i>
<i>branchbyte1</i>
<i>branchbyte2</i>

Forms

ifnull = 198 (0xc6)

Operand Stack

..., *value* $\Rightarrow$...

Description

The *value* must be of type `reference`. It is popped from the operand stack. If *value* is `null`, the *unsigned branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is calculated to be $(branchbyte1 \ll 8) \mid branchbyte2$. Execution then proceeds at that offset from the address of the opcode of this *ifnull* instruction. The target address must be that of an opcode of an instruction within the method that contains this *ifnull* instruction.

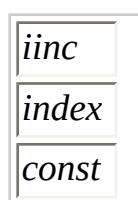
Otherwise, execution proceeds at the address of the instruction following this *ifnull* instruction.

iinc

Operation

Increment local variable by constant

Format



Forms

iinc = 132 (0x84)

Operand Stack

No change

Description

The *index* is an unsigned byte that must be an index into the local variable array of the current frame ([§3.6](#)). The *const* is an immediate signed byte. The local variable at *index* must contain an `int`. The value *const* is first sign-extended to an `int`, and then the local variable at *index* is incremented by that amount.

Notes

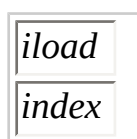
The *iinc* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index and to increment it by a two-byte immediate value.

iload

Operation

Load `int` from local variable

Format



Forms

iload = 21 (0x15)

Operand Stack

... $\Rightarrow$..., *value*

Description

The *index* is an unsigned byte that must be an index into the local variable array of the current frame ([§3.6](#)). The local variable at *index* must contain an `int`. The *value* of the local variable at *index* is pushed onto the operand stack.

Notes

The *iload* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

iload_<n>

Operation

Load `int` from local variable

Format

<i>iload_<n></i>

Forms

iload_0 = 26 (0x1a) *iload_1* = 27 (0x1b) *iload_2* = 28 (0x1c) *iload_3* = 29 (0x1d)

Operand Stack

... $\Rightarrow$..., *value*

Description

The *<n>* must be an index into the local variable array of the current frame ([§3.6](#)). The local variable at *<n>* must contain an `int`. The *value* of the local variable at *<n>* is pushed onto the operand stack.

Notes

Each of the *iload_<n>* instructions is the same as *iload* with an *index* of *<n>*, except that the operand *<n>* is implicit.

imul

Operation

Multiply `int`

Format

<i>imul</i>

Forms

imul = 104 (0x68)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* * *value2*. The *result* is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *imul* instruction never throws a runtime exception.

ineg

Operation

Negate `int`

Format

<i>ineg</i>

Forms

ineg = 116 (0x74)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* must be of type `int`. It is popped from the operand stack. The `int` *result* is the arithmetic negation of *value*, *-value*. The *result* is pushed onto the operand stack.

For `int` values, negation is the same as subtraction from zero. Because the Java virtual machine uses two's-complement representation for integers and the range of two's-complement values is not symmetric, the negation of the maximum negative `int` results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown.

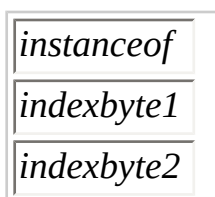
For all `int` values x , $-x$ equals $(\sim x) + 1$.

instanceof

Operation

Determine if object is of given type

Format



Forms

instanceof = 193 (0xc1)

Operand Stack

..., *objectref* $\Rightarrow$..., *result*

Description

The *objectref*, which must be of type `reference`, is popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class ([§3.6](#)), where the value of the index is $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$. The runtime constant pool item at the index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved ([§5.4.3.1](#)).

If *objectref* is not `null` and is an instance of the resolved class or array or implements the resolved interface, the *instanceof* instruction pushes an `int` *result* of 1 as an int on

the operand stack. Otherwise, it pushes an `int` *result* of 0.

The following rules are used to determine whether an *objectref* that is not `null` is an instance of the resolved type: If *S* is the class of the object referred to by *objectref* and *T* is the resolved class, array, or interface type, *instanceof* determines whether *objectref* is an instance of *T* as follows:

- If *S* is an ordinary (nonarray) class, then:
 - If *T* is a class type, then *S* must be the same class ([§2.8.1](#)) as *T* or a subclass of *T*.
 - If *T* is an interface type, then *S* must implement ([§2.13](#)) interface *T*.
- If *S* is an interface type, then:
 - If *T* is a class type, then *T* must be `Object` ([§2.4.7](#)).
 - If *T* is an interface type, then *T* must be the same interface as *S*, or a superinterface of *S* ([§2.13.2](#)).
- If *S* is a class representing the array type `SC[]`, that is, an array of components of type *SC*, then:
 - If *T* is a class type, then *T* must be `Object` ([§2.4.7](#)).
 - If *T* is an array type `TC[]`, that is, an array of components of type *TC*, then one of the following must be true:
 - *TC* and *SC* are the same primitive type ([§2.4.1](#)).
 - *TC* and *SC* are reference types ([§2.4.6](#)), and type *SC* can be cast to *TC* by these runtime rules.
 - If *T* is an interface type, *T* must be one of the interfaces implemented by arrays ([§2.15](#)).

Linking Exceptions

During resolution of symbolic reference to the class, array, or interface type, any of the exceptions documented in [Section 5.4.3.1](#) can be thrown.

Notes

The *instanceof* instruction is very similar to the *checkcast* instruction. It differs in its

treatment of `null`, its behavior when its test fails (*checkcast* throws an exception, *instanceof* pushes a result code), and its effect on the operand stack.

invokeinterface

Operation

Invoke interface method

Format

<i>invokeinterface</i>
<i>indexbyte1</i>
<i>indexbyte2</i>
<i>count</i>
<i>0</i>

Forms

invokeinterface = 185 (0xb9)

Operand Stack

..., *objectref*, [*arg1*, [*arg2* ...]] $\Rightarrow$...

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class ([§3.6](#)), where the value of the index is $(indexbyte1 \ll 8) \mid indexbyte2$. The runtime constant pool item at that index must be a symbolic reference to an interface method ([§5.1](#)), which gives the name and descriptor ([§4.3.3](#)) of the interface method as well as a symbolic reference to the interface in which the interface method is to be found. The named interface method is resolved ([§5.4.3.4](#)). The interface method must not be an instance initialization method ([§3.9](#)) or the class or interface initialization method ([§3.9](#)).

The *count* operand is an unsigned byte that must not be zero. The *objectref* must be of type **reference** and must be followed on the operand stack by *nargs* argument values, where the number, type, and order of the values must be consistent with the

descriptor of the resolved interface method. The value of the fourth operand byte must always be zero.

Let *C* be the class of *objectref*. The actual method to be invoked is selected by the following lookup procedure:

- If *C* contains a declaration for an instance method with the same name and descriptor as the resolved method, then this is the method to be invoked, and the lookup procedure terminates.
- Otherwise, if *C* has a superclass, this same lookup procedure is performed recursively using the direct superclass of *C*; the method to be invoked is the result of the recursive invocation of this lookup procedure.
- Otherwise, an `AbstractMethodError` is raised.

If the method is `synchronized`, the monitor associated with *objectref* is acquired or reentered.

If the method is not `native`, the *nargs* argument values and *objectref* are popped from the operand stack. A new frame is created on the Java virtual machine stack for the method being invoked. The *objectref* and the argument values are consecutively made the values of local variables of the new frame, with *objectref* in local variable 0, *arg1* in local variable 1 (or, if *arg1* is of type `long` or `double`, in local variables 1 and 2), and so on. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java virtual machine `pc` is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is `native` and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java virtual machine, that is done. The *nargs* argument values and *objectref* are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns:

- If the `native` method is `synchronized`, the monitor associated with *objectref* is released or exited as if by execution of a *monitorexit* instruction.
- If the `native` method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the `native` method and pushed onto the operand stack.

Linking Exceptions

During resolution of the symbolic reference to the interface method, any of the exceptions documented in [§5.4.3.4](#) can be thrown.

Runtime Exceptions

Otherwise, if *objectref* is `null`, the *invokeinterface* instruction throws a `NullPointerException`.

Otherwise, if the class of *objectref* does not implement the resolved interface, *invokeinterface* throws an `IncompatibleClassChangeError`.

Otherwise, if no method matching the resolved name and descriptor is selected, *invokeinterface* throws an `AbstractMethodError`.

Otherwise, if the selected method is not `public`, *invokeinterface* throws an `IllegalAccessError`.

Otherwise, if the selected method is `abstract`, *invokeinterface* throws an `AbstractMethodError`.

Otherwise, if the selected method is `native` and the code that implements the method cannot be bound, *invokeinterface* throws an `UnsatisfiedLinkError`.

Notes

The *count* operand of the *invokeinterface* instruction records a measure of the number of argument values, where an argument value of type `long` or type `double` contributes two units to the *count* value and an argument of any other type contributes one unit. This information can also be derived from the descriptor of the selected method. The redundancy is historical.

The fourth operand byte exists to reserve space for an additional operand used in certain of Sun's implementations, which replace the *invokeinterface* instruction by a specialized pseudo-instruction at run time. It must be retained for backwards compatibility.

The *nargs* argument values and *objectref* are not one-to-one with the first *nargs* + 1 local variables. Argument values of types `long` and `double` must be stored in two consecutive local variables, thus more than *nargs* local variables may be required to pass *nargs* argument values to the invoked method.

invokespecial

Operation

Invoke instance method; special handling for superclass, private, and instance initialization method invocations

Format

<i>invokespecial</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

Forms

invokespecial = 183 (0xb7)

Operand Stack

..., *objectref*, [*arg1*, [*arg2* ...]] $\Rightarrow$...

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class ([§3.6](#)), where the value of the index is $(indexbyte1 \ll 8) \mid indexbyte2$. The runtime constant pool item at that index must be a symbolic reference to a method ([§5.1](#)), which gives the name and descriptor ([§4.3.3](#)) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved ([§5.4.3.3](#)). Finally, if the resolved method is **protected** ([§4.6](#)), and it is either a member of the current class or a member of a superclass of the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

Next, the resolved method is selected for invocation unless all of the following conditions are true:

- The ACC_SUPER flag (see [Table 4.1, "Class access and property modifiers"](#)) is set for the current class.
- The class of the resolved method is a superclass of the current class.

- The resolved method is not an instance initialization method ([§3.9](#)).

If the above conditions are true, the actual method to be invoked is selected by the following lookup procedure. Let *C* be the direct superclass of the current class:

- If *C* contains a declaration for an instance method with the same name and descriptor as the resolved method, then this method will be invoked. The lookup procedure terminates.
- Otherwise, if *C* has a superclass, this same lookup procedure is performed recursively using the direct superclass of *C*. The method to be invoked is the result of the recursive invocation of this lookup procedure.
- Otherwise, an `AbstractMethodError` is raised.

The *objectref* must be of type `reference` and must be followed on the operand stack by *nargs* argument values, where the number, type, and order of the values must be consistent with the descriptor of the selected instance method.

If the method is `synchronized`, the monitor associated with *objectref* is acquired or reentered.

If the method is not `native`, the *nargs* argument values and *objectref* are popped from the operand stack. A new frame is created on the Java virtual machine stack for the method being invoked. The *objectref* and the argument values are consecutively made the values of local variables of the new frame, with *objectref* in local variable 0, *arg1* in local variable 1 (or, if *arg1* is of type `long` or `double`, in local variables 1 and 2), and so on. Any argument value that is of a floating-point type undergoes value set conversion ([§3.8.3](#)) prior to being stored in a local variable. The new frame is then made current, and the Java virtual machine `pc` is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is `native` and the platform-dependent code that implements it has not yet been bound ([§5.6](#)) into the Java virtual machine, that is done. The *nargs* argument values and *objectref* are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion ([§3.8.3](#)) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

- If the `native` method is `synchronized`, the monitor associated with *objectref* is released or exited as if by execution of a *monitorexit* instruction.
- If the `native` method returns a value, the return value of the platform-dependent code is

converted in an implementation-dependent way to the return type of the `native` method and pushed onto the operand stack.

Linking Exceptions

During resolution of the symbolic reference to the method, any of the exceptions pertaining to method resolution documented in [Section 5.4.3.3](#) can be thrown.

Otherwise, if the resolved method is an instance initialization method, and the class in which it is declared is not the class symbolically referenced by the instruction, a `NoSuchMethodError` is thrown.

Otherwise, if the resolved method is a class (`static`) method, the *invokespecial* instruction throws an `IncompatibleClassChangeError`.

Otherwise, if no method matching the resolved name and descriptor is selected, *invokespecial* throws an `AbstractMethodError`.

Otherwise, if the selected method is `abstract`, *invokespecial* throws an `AbstractMethodError`.

Runtime Exceptions

Otherwise, if *objectref* is `null`, the *invokespecial* instruction throws a `NullPointerException`.

Otherwise, if the selected method is `native` and the code that implements the method cannot be bound, *invokespecial* throws an `UnsatisfiedLinkError`.

Notes

The difference between the *invokespecial* and the *invokevirtual* instructions is that *invokevirtual* invokes a method based on the class of the object. The *invokespecial* instruction is used to invoke instance initialization methods ([§3.9](#)) as well as `private` methods and methods of a superclass of the current class.

The *invokespecial* instruction was named *invokenonvirtual* prior to Sun's JDK release 1.0.2.

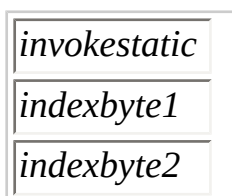
The *nargs* argument values and *objectref* are not one-to-one with the first *nargs* + 1 local variables. Argument values of types `long` and `double` must be stored in two consecutive local variables, thus more than *nargs* local variables may be required to pass *nargs* argument values to the invoked method.

invokestatic

Operation

Invoke a class (`static`) method

Format



Forms

invokestatic = 184 (0xb8)

Operand Stack

..., [*arg1*, [*arg2* ...]] $\Rightarrow$...

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class ([§3.6](#)), where the value of the index is $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$. The runtime constant pool item at that index must be a symbolic reference to a method ([§5.1](#)), which gives the name and descriptor ([§4.3.3](#)) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved ([§5.4.3.3](#)). The method must not be the class or interface initialization method ([§3.9](#)). It must be `static`, and therefore cannot be `abstract`.

On successful resolution of the method, the class that declared the resolved field is initialized ([§5.5](#)) if that class has not already been initialized.

The operand stack must contain *nargs* argument values, where the number, type, and order of the values must be consistent with the descriptor of the resolved method.

If the method is `synchronized`, the monitor associated with the resolved class is acquired or reentered.

If the method is not **native**, the *nargs* argument values are popped from the operand stack. A new frame is created on the Java virtual machine stack for the method being invoked. The *nargs* argument values are consecutively made the values of local variables of the new frame, with *arg1* in local variable 0 (or, if *arg1* is of type **long** or **double**, in local variables 0 and 1) and so on. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java virtual machine **pc** is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is **native** and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java virtual machine, that is done. The *nargs* argument values are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

- If the **native** method is **synchronized**, the monitor associated with the resolved class is released or exited as if by execution of a *monitorexit* instruction.
- If the **native** method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the **native** method and pushed onto the operand stack.

Linking Exceptions

During resolution of the symbolic reference to the method, any of the exceptions pertaining to method resolution documented in [Section 5.4.3.3](#) can be thrown.

Otherwise, if the resolved method is an instance method, the *invokestatic* instruction throws an **IncompatibleClassChangeError**.

Runtime Exceptions

Otherwise, if execution of this *invokestatic* instruction causes initialization of the referenced class, *invokestatic* may throw an **Error** as detailed in [Section 2.17.5](#).

Otherwise, if the resolved method is **native** and the code that implements the method cannot be bound, *invokestatic* throws an **UnsatisfiedLinkError**.

Notes

The *nargs* argument values are not one-to-one with the first *nargs* local variables.

Argument values of types `long` and `double` must be stored in two consecutive local variables, thus more than *nargs* local variables may be required to pass *nargs* argument values to the invoked method.

invokevirtual

Operation

Invoke instance method; dispatch based on class

Format

<i>invokevirtual</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

Forms

invokevirtual = 182 (0xb6)

Operand Stack

..., *objectref*, [*arg1*, [*arg2* ...]] $\Rightarrow$...

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class ([§3.6](#)), where the value of the index is $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$. The runtime constant pool item at that index must be a symbolic reference to a method ([§5.1](#)), which gives the name and descriptor ([§4.3.3](#)) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved ([§5.4.3.3](#)). The method must not be an instance initialization method ([§3.9](#)) or the class or interface initialization method ([§3.9](#)). Finally, if the resolved method is `protected` ([§4.6](#)), and it is either a member of the current class or a member of a superclass of the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

Let C be the class of *objectref*. The actual method to be invoked is selected by the following lookup procedure:

- If *C* contains a declaration for an instance method with the same name and descriptor as the resolved method, and the resolved method is accessible from *C*, then this is the method to be invoked, and the lookup procedure terminates.
- Otherwise, if *C* has a superclass, this same lookup procedure is performed recursively using the direct superclass of *C*; the method to be invoked is the result of the recursive invocation of this lookup procedure.
- Otherwise, an `AbstractMethodError` is raised.

The *objectref* must be followed on the operand stack by *nargs* argument values, where the number, type, and order of the values must be consistent with the descriptor of the selected instance method.

If the method is **synchronized**, the monitor associated with *objectref* is acquired or reentered.

If the method is not **native**, the *nargs* argument values and *objectref* are popped from the operand stack. A new frame is created on the Java virtual machine stack for the method being invoked. The *objectref* and the argument values are consecutively made the values of local variables of the new frame, with *objectref* in local variable 0, *arg1* in local variable 1 (or, if *arg1* is of type **long** or **double**, in local variables 1 and 2), and so on. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java virtual machine **pc** is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is **native** and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java virtual machine, that is done. The *nargs* argument values and *objectref* are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

- If the **native** method is **synchronized**, the monitor associated with *objectref* is released or exited as if by execution of a *monitorexit* instruction.
- If the **native** method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the **native** method and pushed onto the operand stack.

Linking Exceptions

During resolution of the symbolic reference to the method, any of the exceptions pertaining to method resolution documented in [Section 5.4.3.3](#) can be thrown.

Otherwise, if the resolved method is a class (`static`) method, the *invokevirtual* instruction throws an `IncompatibleClassChangeError`.

Runtime Exceptions

Otherwise, if *objectref* is `null`, the *invokevirtual* instruction throws a `NullPointerException`.

Otherwise, if no method matching the resolved name and descriptor is selected, *invokevirtual* throws an `AbstractMethodError`.

Otherwise, if the selected method is `abstract`, *invokevirtual* throws an `AbstractMethodError`.

Otherwise, if the selected method is `native` and the code that implements the method cannot be bound, *invokevirtual* throws an `UnsatisfiedLinkError`.

Notes

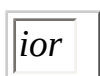
The *nargs* argument values and *objectref* are not one-to-one with the first *nargs* + 1 local variables. Argument values of types `long` and `double` must be stored in two consecutive local variables, thus more than *nargs* local variables may be required to pass *nargs* argument values to the invoked method.

ior

Operation

Boolean OR `int`

Format



Forms

ior = 128 (0x80)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `int`. They are popped from the operand stack. An `int` *result* is calculated by taking the bitwise inclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

irem

Operation

Remainder `int`

Format

<i>irem</i>

Forms

irem = 112 (0x70)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is $value1 - (value1 / value2) * value2$. The *result* is pushed onto the operand stack.

The result of the *irem* instruction is such that $(a/b) * b + (a\%b)$ is equal to *a*. This identity holds even in the special case in which the dividend is the negative `int` of largest possible magnitude for its type and the divisor is `-1` (the remainder is `0`). It follows from this rule that the result of the remainder operation can be negative only if

the dividend is negative and can be positive only if the dividend is positive. Moreover, the magnitude of the result is always less than the magnitude of the divisor.

Runtime Exception

If the value of the divisor for an `int` remainder operator is 0, *irem* throws an `ArithmeticException`.

ireturn

Operation

Return `int` from method

Format

<i>ireturn</i>

Forms

ireturn = 172 (0xac)

Operand Stack

..., *value* $\Rightarrow$ [empty]

Description

The current method must have return type `boolean`, `byte`, `short`, `char`, or `int`. The *value* must be of type `int`. If the current method is a `synchronized` method, the monitor acquired or reentered on invocation of the method is released or exited (respectively) as if by execution of a *monitorexit* instruction. If no exception is thrown, *value* is popped from the operand stack of the current frame ([§3.6](#)) and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

Runtime Exceptions

If the current method is a **synchronized** method and the current thread is not the owner of the monitor acquired or reentered on invocation of the method, *ireturn* throws an `IllegalMonitorStateException`. This can happen, for example, if a **synchronized** method contains a *monitorexit* instruction, but no *monitorenter* instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in [Section 8.13](#) and if the first of those rules is violated during invocation of the current method, then *ireturn* throws an `IllegalMonitorStateException`.

ishl

Operation

Shift left `int`

Format

<i>ishl</i>

Forms

ishl = 120 (0x78)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by shifting *value1* left by *s* bit positions, where *s* is the value of the low 5 bits of *value2*. The *result* is pushed onto the operand stack.

Notes

This is equivalent (even if overflow occurs) to multiplication by 2 to the power *s*. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were

subjected to a bitwise logical AND with the mask value 0x1f.

ishr

Operation

Arithmetic shift right `int`

Format



Forms

ishr = 122 (0x7a)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by shifting *value1* right by *s* bit positions, with sign extension, where *s* is the value of the low 5 bits of *value2*. The *result* is pushed onto the operand stack.

Notes

The resulting value is $\lfloor value1 / 2^s \rfloor$, where *s* is *value2* & 0x1f. For nonnegative *value1*, this is equivalent to truncating `int` division by 2 to the power *s*. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

istore

Operation

Store `int` into local variable

Format



Forms

istore = 54 (0x36)

Operand Stack

..., *value* $\Rightarrow$...

Description

The *index* is an unsigned byte that must be an index into the local variable array of the current frame ([§3.6](#)). The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack, and the value of the local variable at *index* is set to *value*.

Notes

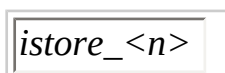
The *istore* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

`istore_<n>`

Operation

Store `int` into local variable

Format



Forms

istore_0 = 59 (0x3b) *istore_1* = 60 (0x3c) *istore_2* = 61 (0x3d) *istore_3* = 62 (0x3e)

Operand Stack

..., *value* $\Rightarrow$...

Description

The $\langle n \rangle$ must be an index into the local variable array of the current frame ([§3.6](#)). The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack, and the value of the local variable at $\langle n \rangle$ is set to *value*.

Notes

Each of the *istore_* $\langle n \rangle$ instructions is the same as *istore* with an *index* of $\langle n \rangle$, except that the operand $\langle n \rangle$ is implicit.

isub

Operation

Subtract `int`

Format

<i>isub</i>

Forms

isub = 100 (0x64)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand

stack. The *int result* is *value1* - *value2*. The *result* is pushed onto the operand stack.

For *int* subtraction, *a* - *b* produces the same result as *a* + (-*b*). For *int* values, subtraction from zero is the same as negation.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type *int*. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *isub* instruction never throws a runtime exception.

iushr

Operation

Logical shift right *int*

Format



Forms

iushr = 124 (0x7c)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type *int*. The values are popped from the operand stack. An *int result* is calculated by shifting *value1* right by *s* bit positions, with zero extension, where *s* is the value of the low 5 bits of *value2*. The *result* is pushed onto the operand stack.

Notes

If *value1* is positive and *s* is *value2* & 0x1f, the result is the same as that of *value1* >> *s*; if *value1* is negative, the result is equal to the value of the expression (*value1* >> *s*) + (2 << ~*s*). The addition of the (2 << ~*s*) term cancels out the propagated sign bit. The shift distance actually used is always in the range 0 to 31, inclusive.

ixor

Operation

Boolean XOR `int`

Format



Forms

ixor = 130 (0x82)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `int`. They are popped from the operand stack. An `int` *result* is calculated by taking the bitwise exclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The JavaTM Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections to jvm@java.sun.com

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

l2d

Operation

Convert `long` to `double`

Format

<i>l2d</i>

Forms

l2d = 138 (0x8a)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* on the top of the operand stack must be of type `long`. It is popped from the operand stack and converted to a `double` *result* using IEEE 754 round to nearest mode. The *result* is pushed onto the operand stack.

Notes

The *l2d* instruction performs a widening primitive conversion ([§2.6.2](#)) that may lose precision because values of type `double` have only 53 significand bits.

l2f

Operation

Convert `long` to `float`

Format

<i>l2f</i>

Forms

l2f = 137 (0x89)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* on the top of the operand stack must be of type `long`. It is popped from the operand stack and converted to a `float` *result* using IEEE 754 round to nearest mode. The *result* is pushed onto the operand stack.

Notes

The *l2f* instruction performs a widening primitive conversion ([§2.6.2](#)) that may lose precision because values of type `float` have only 24 significand bits.

l2i

Operation

Convert `long` to `int`

Format

<i>l2i</i>

Forms

l2i = 136 (0x88)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* on the top of the operand stack must be of type `long`. It is popped from the operand stack and converted to an `int` *result* by taking the low-order 32 bits of the `long` value and discarding the high-order 32 bits. The *result* is pushed onto the operand stack.

Notes

The *l2i* instruction performs a narrowing primitive conversion ([§2.6.3](#)). It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign as *value*.

ladd

Operation

Add long

Format

<i>ladd</i>

Forms

ladd = 97 (0x61)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `long`. The values are popped from the operand stack. The `long` *result* is *value1* + *value2*. The *result* is pushed onto the

operand stack.

The result is the 64 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `long`. If overflow occurs, the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *ladd* instruction never throws a runtime exception.

laload

Operation

Load `long` from array

Format

<i>laload</i>

Forms

laload = 47 (0x2f)

Operand Stack

..., *arrayref*, *index* $\Rightarrow$..., *value*

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `long`. The *index* must be of type `int`. Both *arrayref* and *index* are popped from the operand stack. The `long` *value* in the component of the array at *index* is retrieved and pushed onto the operand stack.

Runtime Exceptions

If *arrayref* is `null`, *laload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the

laload instruction throws an `ArrayIndexOutOfBoundsException`.

land

Operation

Boolean AND `long`

Format

<i>land</i>

Forms

land = 127 (0x7f)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `long`. They are popped from the operand stack. A `long` *result* is calculated by taking the bitwise AND of *value1* and *value2*. The *result* is pushed onto the operand stack.

lastore

Operation

Store into `long` array

Format

<i>lastore</i>

Forms

lastore = 80 (0x50)

Operand Stack

..., *arrayref*, *index*, *value* $\Rightarrow$...

Description

The *arrayref* must be of type **reference** and must refer to an array whose components are of type **long**. The *index* must be of type **int**, and *value* must be of type **long**. The *arrayref*, *index*, and *value* are popped from the operand stack. The **long** *value* is stored as the component of the array indexed by *index*.

Runtime Exceptions

If *arrayref* is **null**, *lastore* throws a **NullPointerException**.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *lastore* instruction throws an **ArrayIndexOutOfBoundsException**.

lcmp

Operation

Compare **long**

Format



Forms

lcmp = 148 (0x94)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `long`. They are both popped from the operand stack, and a signed integer comparison is performed. If *value1* is greater than *value2*, the `int` value 1 is pushed onto the operand stack. If *value1* is equal to *value2*, the `int` value 0 is pushed onto the operand stack. If *value1* is less than *value2*, the `int` value -1 is pushed onto the operand stack.

`lconst_<l>`

Operation

Push `long` constant

Format

<code>lconst_<l></code>

Forms

`lconst_0` = 9 (0x9) `lconst_1` = 10 (0xa)

Operand Stack

`...` $\Rightarrow$ `..., <l>`

Description

Push the `long` constant `<l>` (0 or 1) onto the operand stack.

`ldc`

Operation

Push item from runtime constant pool

Format



Forms

$ldc = 18$ (0x12)

Operand Stack

$\dots \Rightarrow \dots, value$

Description

The *index* is an unsigned byte that must be a valid index into the runtime constant pool of the current class (§3.6). The runtime constant pool entry at *index* either must be a runtime constant of type `int` or `float`, or must be a symbolic reference to a string literal (§5.1).

If the runtime constant pool entry is a runtime constant of type `int` or `float`, the numeric *value* of that runtime constant is pushed onto the operand stack as an `int` or `float`, respectively.

Otherwise, the runtime constant pool entry must be a reference to an instance of class `String` representing a string literal (§5.1). A *reference* to that instance, *value*, is pushed onto the operand stack.

Notes

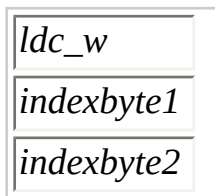
The *ldc* instruction can only be used to push a value of type `float` taken from the float value set (§3.3.2) because a constant of type `float` in the constant pool (§4.4.4) must be taken from the float value set.

ldc_w

Operation

Push item from runtime constant pool (wide index)

Format



Forms

ldc_w = 19 (0x13)

Operand Stack

... $\Rightarrow$..., *value*

Description

The unsigned *indexbyte1* and *indexbyte2* are assembled into an unsigned 16-bit index into the runtime constant pool of the current class ([§3.6](#)), where the value of the index is calculated as $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$. The index must be a valid index into the runtime constant pool of the current class. The runtime constant pool entry at the index either must be a runtime constant of type `int` or `float`, or must be a symbolic reference to a string literal ([§5.1](#)).

If the runtime constant pool entry is a runtime constant of type `int` or `float`, the numeric *value* of that runtime constant is pushed onto the operand stack as an `int` or `float`, respectively.

Otherwise, the runtime constant pool entry must be a reference to an instance of class `String` representing a string literal ([§5.1](#)). A reference to that instance, *value*, is pushed onto the operand stack.

Notes

The *ldc_w* instruction is identical to the *ldc* instruction except for its wider runtime constant pool index.

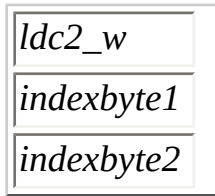
The *ldc_w* instruction can only be used to push a value of type `float` taken from the float value set ([§3.3.2](#)) because a constant of type `float` in the constant pool ([§4.4.4](#)) must be taken from the float value set.

ldc2_w

Operation

Push `long` or `double` from runtime constant pool (wide index)

Format



Forms

ldc2_w = 20 (0x14)

Operand Stack

... $\Rightarrow$..., *value*

Description

The unsigned *indexbyte1* and *indexbyte2* are assembled into an unsigned 16-bit index into the runtime constant pool of the current class ([§3.6](#)), where the value of the index is calculated as $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$. The index must be a valid index into the runtime constant pool of the current class. The runtime constant pool entry at the index must be a runtime constant of type `long` or `double` ([§5.1](#)). The numeric *value* of that runtime constant is pushed onto the operand stack as a `long` or `double`, respectively.

Notes

Only a wide-index version of the *ldc2_w* instruction exists; there is no *ldc2* instruction that pushes a `long` or `double` with a single-byte index.

The *ldc2_w* instruction can only be used to push a value of type `double` taken from the double value set ([§3.3.2](#)) because a constant of type `double` in the constant pool ([§4.4.5](#)) must be taken from the double value set.

ldiv

Operation

Divide `long`

Format

`ldiv`

Forms

`ldiv` = 109 (0x6d)

Operand Stack

$\dots, value1, value2 \Rightarrow \dots, result$

Description

Both *value1* and *value2* must be of type `long`. The values are popped from the operand stack. The `long result` is the value of the Java programming language expression *value1* / *value2*. The *result* is pushed onto the operand stack.

A `long` division rounds towards 0; that is, the quotient produced for `long` values in n / d is a `long` value q whose magnitude is as large as possible while satisfying $|d \cdot q| \leq |n|$. Moreover, q is positive when $|n| \geq |d|$ and n and d have the same sign, but q is negative when $|n| \geq |d|$ and n and d have opposite signs.

There is one special case that does not satisfy this rule: if the dividend is the negative integer of largest possible magnitude for the `long` type and the divisor is -1, then overflow occurs and the result is equal to the dividend; despite the overflow, no exception is thrown in this case.

Runtime Exception

If the value of the divisor in a `long` division is 0, `ldiv` throws an `ArithmeticException`.

lload

Operation

Load long from local variable

Format



Forms

lload = 22 (0x16)

Operand Stack

... $\Rightarrow$..., *value*

Description

The *index* is an unsigned byte. Both *index* and *index* + 1 must be indices into the local variable array of the current frame ([§3.6](#)). The local variable at *index* must contain a long. The *value* of the local variable at *index* is pushed onto the operand stack.

Notes

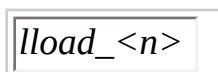
The *lload* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

lload_<n>

Operation

Load long from local variable

Format



Forms

lload_0 = 30 (0x1e) *lload_1* = 31 (0x1f) *lload_2* = 32 (0x20) *lload_3* = 33 (0x21)

Operand Stack

... $\Rightarrow$..., *value*

Description

Both $\langle n \rangle$ and $\langle n \rangle + 1$ must be indices into the local variable array of the current frame ([§3.6](#)). The local variable at $\langle n \rangle$ must contain a **long**. The *value* of the local variable at $\langle n \rangle$ is pushed onto the operand stack.

Notes

Each of the *lload_* $\langle n \rangle$ instructions is the same as *lload* with an *index* of $\langle n \rangle$, except that the operand $\langle n \rangle$ is implicit.

Imul

Operation

Multiply long

Format

Imul

Forms

Imul = 105 (0x69)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type **long**. The values are popped from the

operand stack. The **long** *result* is *value1* * *value2*. The *result* is pushed onto the operand stack.

The result is the 64 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type **long**. If overflow occurs, the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *lmul* instruction never throws a runtime exception.

Ineg

Operation

Negate **long**

Format



Forms

lneg = 117 (0x75)

Operand Stack

..., *value* $\Rightarrow$..., *result*

Description

The *value* must be of type **long**. It is popped from the operand stack. The **long** *result* is the arithmetic negation of *value*, -*value*. The *result* is pushed onto the operand stack.

For **long** values, negation is the same as subtraction from zero. Because the Java virtual machine uses two's-complement representation for integers and the range of two's-complement values is not symmetric, the negation of the maximum negative **long** results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown.

For all `long` values x , $-x$ equals $(\sim x) + 1$.

lookupswitch

Operation

Access jump table by key match and jump

Format

<i>lookupswitch</i>
<i><0-3 byte pad></i>
<i>defaultbyte1</i>
<i>defaultbyte2</i>
<i>defaultbyte3</i>
<i>defaultbyte4</i>
<i>npairs1</i>
<i>npairs2</i>
<i>npairs3</i>
<i>npairs4</i>
<i>match-offset</i>
<i>pairs...</i>

Forms

lookupswitch = 171 (0xab)

Operand Stack

..., *key* $\Rightarrow$...

Description

A *lookupswitch* is a variable-length instruction. Immediately after the *lookupswitch* opcode, between zero and three null bytes (zeroed bytes, not the null object) are inserted as padding. The number of null bytes is chosen so that the *defaultbyte1* begins at an address that is a multiple of four bytes from the start of the current method (the

opcode of its first instruction). Immediately after the padding follow a series of signed 32-bit values: *default*, *npairs*, and then *npairs* pairs of signed 32-bit values. The *npairs* must be greater than or equal to 0. Each of the *npairs* pairs consists of an `int` *match* and a signed 32-bit *offset*. Each of these signed 32-bit values is constructed from four unsigned bytes as $(\text{byte1} \ll 24) \mid (\text{byte2} \ll 16) \mid (\text{byte3} \ll 8) \mid \text{byte4}$.

The table *match-offset* pairs of the *lookupswitch* instruction must be sorted in increasing numerical order by *match*.

The *key* must be of type `int` and is popped from the operand stack. The *key* is compared against the *match* values. If it is equal to one of them, then a target address is calculated by adding the corresponding *offset* to the address of the opcode of this *lookupswitch* instruction. If the *key* does not match any of the *match* values, the target address is calculated by adding *default* to the address of the opcode of this *lookupswitch* instruction. Execution then continues at the target address.

The target address that can be calculated from the offset of each *match-offset* pair, as well as the one calculated from *default*, must be the address of an opcode of an instruction within the method that contains this *lookupswitch* instruction.

Notes

The alignment required of the 4-byte operands of the *lookupswitch* instruction guarantees 4-byte alignment of those operands if and only if the method that contains the *lookupswitch* is positioned on a 4-byte boundary.

The *match-offset* pairs are sorted to support lookup routines that are quicker than linear search.

lor

Operation

Boolean OR long

Format



Forms

lor = 129 (0x81)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type **long**. They are popped from the operand stack. A **long** *result* is calculated by taking the bitwise inclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

lrem

Operation

Remainder **long**

Format

<i>lrem</i>

Forms

lrem = 113 (0x71)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type **long**. The values are popped from the operand stack. The **long** *result* is *value1* - (*value1* / *value2*) * *value2*. The *result* is pushed onto the operand stack.

The result of the *lrem* instruction is such that $(a/b)*b + (a\%b)$ is equal to *a*. This identity holds even in the special case in which the dividend is the negative **long** of largest possible magnitude for its type and the divisor is -1 (the remainder is 0). It follows from this rule that the result of the remainder operation can be negative only if

the dividend is negative and can be positive only if the dividend is positive; moreover, the magnitude of the result is always less than the magnitude of the divisor.

Runtime Exception

If the value of the divisor for a `long` remainder operator is 0, *lrem* throws an `ArithmeticException`.

lreturn

Operation

Return `long` from method

Format

<i>lreturn</i>

Forms

lreturn = 173 (0xad)

Operand Stack

..., *value* $\Rightarrow$ [empty]

Description

The current method must have return type `long`. The *value* must be of type `long`. If the current method is a **synchronized** method, the monitor acquired or reentered on invocation of the method is released or exited (respectively) as if by execution of a *monitorexit* instruction. If no exception is thrown, *value* is popped from the operand stack of the current frame ([§3.6](#)) and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

Runtime Exceptions

If the current method is a **synchronized** method and the current thread is not the owner of the monitor acquired or reentered on invocation of the method, *lreturn* throws an `IllegalMonitorStateException`. This can happen, for example, if a **synchronized** method contains a *monitorexit* instruction, but no *monitorenter* instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in [Section 8.13](#) and if the first of those rules is violated during invocation of the current method, then *lreturn* throws an `IllegalMonitorStateException`.

lshl

Operation

Shift left

Format



long

Forms

lshl = 121 (0x79)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

The *value1* must be of type `long`, and *value2* must be of type `int`. The values are popped from the operand stack. A `long result` is calculated by shifting *value1* left by *s* bit positions, where *s* is the low 6 bits of *value2*. The *result* is pushed onto the operand stack.

Notes

This is equivalent (even if overflow occurs) to multiplication by 2 to the power *s*. The shift distance actually used is therefore always in the range 0 to 63, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x3f.

lshr

Operation

Arithmetic shift right **long**

Format



Forms

lshr = 123 (0x7b)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

The *value1* must be of type **long**, and *value2* must be of type **int**. The values are popped from the operand stack. A **long** *result* is calculated by shifting *value1* right by *s* bit positions, with sign extension, where *s* is the value of the low 6 bits of *value2*. The *result* is pushed onto the operand stack.

Notes

The resulting value is $\lfloor value1 / 2^s \rfloor$, where *s* is *value2* & 0x3f. For nonnegative *value1*, this is equivalent to truncating **long** division by 2 to the power *s*. The shift distance actually used is therefore always in the range 0 to 63, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x3f.

lstore

Operation

Store `long` into local variable

Format



Forms

lstore = 55 (0x37)

Operand Stack

..., *value* $\Rightarrow$...

Description

The *index* is an unsigned byte. Both *index* and *index* + 1 must be indices into the local variable array of the current frame ([§3.6](#)). The *value* on the top of the operand stack must be of type `long`. It is popped from the operand stack, and the local variables at *index* and *index* + 1 are set to *value*.

Notes

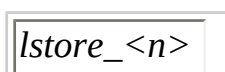
The *lstore* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

lstore_<n>

Operation

Store `long` into local variable

Format



Forms

lstore_0 = 63 (0x3f) *lstore_1* = 64 (0x40) *lstore_2* = 65 (0x41) *lstore_3* = 66 (0x42)

Operand Stack

..., *value* $\Rightarrow$...

Description

Both $\langle n \rangle$ and $\langle n \rangle + 1$ must be indices into the local variable array of the current frame ([§3.6](#)). The *value* on the top of the operand stack must be of type `long`. It is popped from the operand stack, and the local variables at $\langle n \rangle$ and $\langle n \rangle + 1$ are set to *value*.

Notes

Each of the *lstore_* $\langle n \rangle$ instructions is the same as *lstore* with an *index* of $\langle n \rangle$, except that the operand $\langle n \rangle$ is implicit.

lsub

Operation

Subtract `long`

Format

<i>lsub</i>

Forms

lsub = 101 (0x65)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type **long**. The values are popped from the operand stack. The **long result** is *value1* - *value2*. The *result* is pushed onto the operand stack.

For **long** subtraction, **a-b** produces the same result as **a+(-b)**. For **long** values, subtraction from zero is the same as negation.

The result is the 64 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type **long**. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *lsub* instruction never throws a runtime exception.

lushr

Operation

Logical shift right **long**

Format



Forms

lushr = 125 (0x7d)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

The *value1* must be of type **long**, and *value2* must be of type **int**. The values are popped from the operand stack. A **long result** is calculated by shifting *value1* right logically (with zero extension) by the amount indicated by the low 6 bits of *value2*. The *result* is pushed onto the operand stack.

Notes

If *value1* is positive and *s* is *value2* & 0x3f, the result is the same as that of *value1* >> *s*; if *value1* is negative, the result is equal to the value of the expression (*value1* >> *s*) + (2L << ~*s*). The addition of the (2L << ~*s*) term cancels out the propagated sign bit. The shift distance actually used is always in the range 0 to 63, inclusive.

lxor

Operation

Boolean XOR long

Format

<i>lxor</i>

Forms

lxor = 131 (0x83)

Operand Stack

..., *value1*, *value2* $\Rightarrow$..., *result*

Description

Both *value1* and *value2* must be of type `long`. They are popped from the operand stack. A `long` *result* is calculated by taking the bitwise exclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The JavaTM Virtual Machine Specification

Copyright © 1999 Sun Microsystems, Inc. All rights reserved

Please send any comments or corrections to jvm@java.sun.com

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

baload

Operation

Load `byte` or `boolean` from array

Format

<i>baload</i>

Forms

baload = 51 (0x33)

Operand Stack

..., *arrayref*, *index* $\Rightarrow$..., *value*

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `byte` or of type `boolean`. The *index* must be of type `int`. Both *arrayref* and *index* are popped from the operand stack. If the components of the array are of type `byte`, the component of the array at *index* is retrieved and sign-extended to an `int` *value*. If the components of the array are of type `boolean`, the component of the array at *index* is retrieved and zero-extended to an `int` *value*. In either case the resulting *value* is pushed onto the operand stack.

Runtime Exceptions

If *arrayref* is `null`, *baload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *baload* instruction throws an `ArrayIndexOutOfBoundsException`.

Notes

The *baload* instruction is used to load values from both `byte` and `boolean` arrays. In Sun's implementation of the Java virtual machine, `boolean` arrays (arrays of type `T_BOOLEAN`; see [§3.2](#) and the description of the *newarray* instruction in this chapter) are implemented as arrays of 8-bit values. Other implementations may implement packed `boolean` arrays; the *baload* instruction of such implementations must be used to access those arrays.

bastore

Operation

Store into `byte` or `boolean` array

Format

<i>bastore</i>

Forms

bastore = 84 (0x54)

Operand Stack

..., *arrayref*, *index*, *value* $\Rightarrow$...

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `byte` or of type `boolean`. The *index* and the *value* must both be of type `int`. The *arrayref*, *index*, and *value* are popped from the operand stack. If the components of the array are of type `byte`, the `int` *value* is truncated to a `byte` and stored as the component of the array indexed by *index*. If the components of the array are of type `boolean`, the `int` *value* is truncated to its low order bit then zero-extended to the storage size for components of `boolean` arrays used by the implementation. The result is stored as the component of the array indexed by *index*.

Runtime Exceptions

If *arrayref* is `null`, *bastore* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *bastore* instruction throws an `ArrayIndexOutOfBoundsException`.

Notes

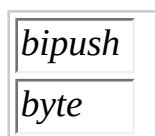
The *bastore* instruction is used to store values into both `byte` and `boolean` arrays. In Sun's implementation of the Java virtual machine, `boolean` arrays (arrays of type `T_BOOLEAN`; see §3.2 and the description of the *newarray* instruction in this chapter) are implemented as arrays of 8-bit values. Other implementations may implement packed `boolean` arrays; in such implementations the *bastore* instruction must be able to store `boolean` values into packed `boolean` arrays as well as `byte` values into `byte` arrays.

bipush

Operation

Push byte

Format



Forms

bipush = 16 (0x10)

Operand Stack

... $\Rightarrow$..., *value*

Description

The immediate *byte* is sign-extended to an `int` *value*. That *value* is pushed onto the operand stack.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The JavaTM Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections to jvm@java.sun.com

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

pop

Operation

Pop the top operand stack value

Format

<i>pop</i>

Forms

pop = 87 (0x57)

Operand Stack

..., *value* $\Rightarrow$...

Description

Pop the top value from the operand stack.

The *pop* instruction must not be used unless *value* is a value of a category 1 computational type ([§3.11.1](#)).

pop2

Operation

Pop the top one or two operand stack values

Format

<i>pop2</i>

Forms

pop2 = 88 (0x58)

Operand Stack

Form 1:

..., *value2*, *value1* $\Rightarrow$...

where each of *value1* and *value2* is a value of a category 1 computational type ([§3.11.1](#)).

Form 2:

..., *value* $\Rightarrow$...

where *value* is a value of a category 2 computational type ([§3.11.1](#)).

Description

Pop the top one or two values from the operand stack.

putfield

Operation

Set field in object

Format

<i>putfield</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

Forms

putfield = 181 (0xb5)

Operand Stack

..., *objectref*, *value* $\Rightarrow$...

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$. The runtime constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The class of *objectref* must not be an array. If the field is **protected** (§4.6), and it is either a member of the current class or a member of a superclass of the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

The referenced field is resolved (§5.4.3.2). The type of a *value* stored by a *putfield* instruction must be compatible with the descriptor of the referenced field (§4.3.2). If the field descriptor type is **boolean**, **byte**, **char**, **short**, or **int**, then the *value* must be an **int**. If the field descriptor type is **float**, **long**, or **double**, then the *value* must be a **float**, **long**, or **double**, respectively. If the field descriptor type is a reference type, then the *value* must be of a type that is assignment compatible (§2.6.7) with the field descriptor type. If the field is **final**, it should be declared in the current class. Otherwise, an `IllegalAccessException` is thrown.

The *value* and *objectref* are popped from the operand stack. The *objectref* must be of type **reference**. The *value* undergoes value set conversion (§3.8.3), resulting in *value'*, and the referenced field in *objectref* is set to *value'*.

Linking Exceptions

During resolution of the symbolic reference to the field, any of the exceptions pertaining to field resolution documented in [Section 5.4.3.2](#) can be thrown.

Otherwise, if the resolved field is a **static** field, *putfield* throws an `IncompatibleClassChangeError`.

Otherwise, if the field is **final**, it must be declared in the current class. Otherwise, an `IllegalAccessException` is thrown.

Runtime Exception

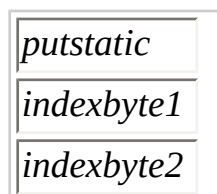
Otherwise, if *objectref* is `null`, the *putfield* instruction throws a `NullPointerException`.

putstatic

Operation

Set `static` field in class

Format



Forms

putstatic = 179 (0xb3)

Operand Stack

..., *value* $\Rightarrow$...

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class ([§3.6](#)), where the value of the index is $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$. The runtime constant pool item at that index must be a symbolic reference to a field ([§5.1](#)), which gives the name and descriptor of the field as well as a symbolic reference to the class or interface in which the field is to be found. The referenced field is resolved ([§5.4.3.2](#)).

On successful resolution of the field the class or interface that declared the resolved field is initialized ([§5.5](#)) if that class or interface has not already been initialized.

The type of a *value* stored by a *putstatic* instruction must be compatible with the descriptor of the referenced field ([§4.3.2](#)). If the field descriptor type is `boolean`,

byte, char, short, or int, then the *value* must be an int. If the field descriptor type is float, long, or double, then the *value* must be a float, long, or double, respectively. If the field descriptor type is a reference type, then the *value* must be of a type that is assignment compatible (§2.6.7) with the field descriptor type. If the field is final, it should be declared in the current class. Otherwise, an `IllegalAccessError` is thrown.

The *value* is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in *value'*. The class field is set to *value'*.

Linking Exceptions

During resolution of the symbolic reference to the class or interface field, any of the exceptions pertaining to field resolution documented in [Section 5.4.3.2](#) can be thrown.

Otherwise, if the resolved field is not a static (class) field or an interface field, *putstatic* throws an `IncompatibleClassChangeError`.

Otherwise, if the field is final, it must be declared in the current class. Otherwise, an `IllegalAccessError` is thrown.

Runtime Exception

Otherwise, if execution of this *putstatic* instruction causes initialization of the referenced class or interface, *putstatic* may throw an `Error` as detailed in [Section 2.17.5](#).

Notes

A *putstatic* instruction may be used only to set the value of an interface field on the initialization of that field. Interface fields may be assigned to only once, on execution of an interface variable initialization expression when the interface is initialized (§2.17.4).

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The Java™ Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections to jvm@java.sun.com

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

saload

Operation

Load `short` from array

Format

<i>saload</i>

Forms

saload = 53 (0x35)

Operand Stack

..., *arrayref*, *index* $\Rightarrow$..., *value*

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `short`. The *index* must be of type `int`. Both *arrayref* and *index* are popped from the operand stack. The component of the array at *index* is retrieved and sign-extended to an `int` *value*. That *value* is pushed onto the operand stack.

Runtime Exceptions

If *arrayref* is `null`, *saload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *saload* instruction throws an `ArrayIndexOutOfBoundsException`.

sastore

Operation

Store into `short` array

Format

<i>sastore</i>

Forms

sastore = 86 (0x56)

Operand Stack

..., *array*, *index*, *value* $\Rightarrow$...

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `short`. Both *index* and *value* must be of type `int`. The *arrayref*, *index*, and *value* are popped from the operand stack. The `int` *value* is truncated to a `short` and stored as the component of the array indexed by *index*.

Runtime Exceptions

If *arrayref* is `null`, *sastore* throws a `NullPointerException`.

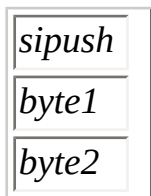
Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *sastore* instruction throws an `ArrayIndexOutOfBoundsException`.

sipush

Operation

Push `short`

Format



Forms

sipush = 17 (0x11)

Operand Stack

... $\Rightarrow$..., *value*

Description

The immediate unsigned *byte1* and *byte2* values are assembled into an intermediate `short` where the value of the short is $(\text{byte1} \ll 8) \mid \text{byte2}$. The intermediate value is then sign-extended to an `int` *value*. That *value* is pushed onto the operand stack.

swap

Operation

Swap the top two operand stack values

Format



Forms

swap = 95 (0x5f)

Operand Stack

..., *value2*, *value1* $\Rightarrow$..., *value1*, *value2*

Description

Swap the top two values on the operand stack.

The *swap* instruction must not be used unless *value1* and *value2* are both values of a category 1 computational type ([§3.11.1](#)).

Notes

The Java virtual machine does not provide an instruction implementing a swap on operands of category 2 computational types.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The JavaTM Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections to jvm@java.sun.com

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

getfield

Operation

Fetch field from object

Format

<i>getfield</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

Forms

getfield = 180 (0xb4)

Operand Stack

..., *objectref* $\Rightarrow$..., *value*

Description

The *objectref*, which must be of type **reference**, is popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$. The runtime constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The referenced field is resolved (§5.4.3.2). The *value* of the referenced field in *objectref* is fetched and pushed onto the operand stack.

The class of *objectref* must not be an array. If the field is **protected** (§4.6), and it is either a member of the current class or a member of a superclass of the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

Linking Exceptions

During resolution of the symbolic reference to the field, any of the errors pertaining to field resolution documented in [Section 5.4.3.2](#) can be thrown.

Otherwise, if the resolved field is a `static` field, *getfield* throws an `IncompatibleClassChangeError`.

Runtime Exception

Otherwise, if *objectref* is `null`, the *getfield* instruction throws a `NullPointerException`.

Notes

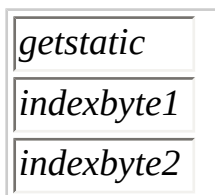
The *getfield* instruction cannot be used to access the `length` field of an array. The *arraylength* instruction is used instead.

getstatic

Operation

Get `static` field from class

Format



Forms

getstatic = 178 (0xb2)

Operand Stack

..., $\Rightarrow$..., *value*

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class ([§3.6](#)), where the value of the index is $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$. The runtime constant pool item at that index must be a symbolic reference to a field ([§5.1](#)), which gives the name and descriptor of the field as well as a symbolic reference to the class or interface in which the field is to be found. The referenced field is resolved ([§5.4.3.2](#)).

On successful resolution of the field, the class or interface that declared the resolved field is initialized ([§5.5](#)) if that class or interface has not already been initialized.

The *value* of the class or interface field is fetched and pushed onto the operand stack.

Linking Exceptions

During resolution of the symbolic reference to the class or interface field, any of the exceptions pertaining to field resolution documented in [Section 5.4.3.2](#) can be thrown.

Otherwise, if the resolved field is not a `static` (class) field or an interface field, *getstatic* throws an `IncompatibleClassChangeError`.

Runtime Exception

Otherwise, if execution of this *getstatic* instruction causes initialization of the referenced class or interface, *getstatic* may throw an `Error` as detailed in [Section 2.17.5](#).

goto

Operation

Branch always

Format

<i>goto</i>
<i>branchbyte1</i>
<i>branchbyte2</i>

Forms

goto = 167 (0xa7)

Operand Stack

No change

Description

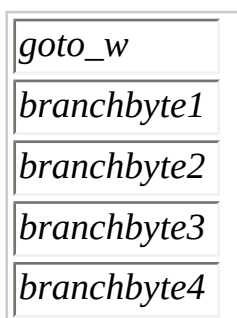
The unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is $(branchbyte1 \ll 8) \mid branchbyte2$. Execution proceeds at that offset from the address of the opcode of this *goto* instruction. The target address must be that of an opcode of an instruction within the method that contains this *goto* instruction.

goto_w

Operation

Branch always (wide index)

Format



Forms

goto_w = 200 (0xc8)

Operand Stack

No change

Description

The unsigned bytes *branchbyte1*, *branchbyte2*, *branchbyte3*, and *branchbyte4* are used to construct a signed 32-bit *branchoffset*, where *branchoffset* is $(branchbyte1 \ll 24) \mid (branchbyte2 \ll 16) \mid (branchbyte3 \ll 8) \mid branchbyte4$. Execution proceeds at that offset from the address of the opcode of this *goto_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *goto_w* instruction.

Notes

Although the *goto_w* instruction takes a 4-byte branch offset, other factors limit the size of a method to 65535 bytes ([§4.10](#)). This limit may be raised in a future release of the Java virtual machine.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The JavaTM Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections to jvm@java.sun.com

CHAPTER 4

The `class` File Format

This chapter describes the Java virtual machine `class` file format. Each `class` file contains the definition of a single class or interface. Although a class or interface need not have an external representation literally contained in a file (for instance, because the class is generated by a class loader), we will colloquially refer to any valid representation of a class or interface as being in the `class` file format.

A `class` file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, and 64-bit quantities are constructed by reading in two, four, and eight consecutive 8-bit bytes, respectively. Multibyte data items are always stored in big-endian order, where the high bytes come first. In the Java and Java 2 platforms, this format is supported by interfaces `java.io.DataInput` and `java.io.DataOutput` and classes such as `java.io.DataInputStream` and `java.io.DataOutputStream`.

This chapter defines its own set of data types representing `class` file data: The types `u1`, `u2`, and `u4` represent an unsigned one-, two-, or four-byte quantity, respectively. In the Java and Java 2 platforms, these types may be read by methods such as `readUnsignedByte`, `readUnsignedShort`, and `readInt` of the interface `java.io.DataInput`.

This chapter presents the `class` file format using pseudostructures written in a C-like structure notation. To avoid confusion with the fields of classes and class instances, etc., the contents of the structures describing the `class` file format are referred to as *items*. Successive items are stored in the `class` file sequentially, without padding or alignment.

Tables, consisting of zero or more variable-sized items, are used in several `class` file structures. Although we use C-like array syntax to refer to table items, the fact that tables are streams of varying-sized structures means that it is not possible to translate a table index directly to a byte offset into the table.

Where we refer to a data structure as an array, it consists of zero or more contiguous fixed-sized items and can be indexed like an array.

4.1 The ClassFile Structure

A `class` file consists of a single `ClassFile` structure:

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

The items in the `ClassFile` structure are as follows:

`magic`

The `magic` item supplies the magic number identifying the `class` file format; it has the value `0xCAFEBAFE`.

`minor_version, major_version`

The values of the `minor_version` and `major_version` items are the minor and major version numbers of this `class` file. Together, a major and a minor version number determine the version of the `class` file format. If a `class` file has major version number M and minor version number m , we denote the version of its `class` file format as $M.m$. Thus, `class` file format versions may be ordered lexicographically, for example, $1.5 < 2.0 < 2.1$.

A Java virtual machine implementation can support a `class` file format of version v if and only if v lies in some contiguous range $M_i.0 \leq v \leq M_j.m$. Only Sun can specify what range of versions a Java virtual machine implementation conforming to a certain release level of the Java platform may support.^{[1](#)}

`constant_pool_count`

The value of the `constant_pool_count` item is equal to the number of entries in the `constant_pool` table plus one. A `constant_pool` index is considered valid if it is

greater than zero and less than `constant_pool_count`, with the exception for constants of type `long` and `double` noted in [§4.4.5](#).

`constant_pool[]`

The `constant_pool` is a table of structures ([§4.4](#)) representing various string constants, class and interface names, field names, and other constants that are referred to within the `ClassFile` structure and its substructures. The format of each `constant_pool` table entry is indicated by its first "tag" byte.

The `constant_pool` table is indexed from 1 to `constant_pool_count-1`.

`access_flags`

The value of the `access_flags` item is a mask of flags used to denote access permissions to and properties of this class or interface. The interpretation of each flag, when set, is as shown in [Table 4.1](#).

Flag Name	Value	Interpretation
<code>ACC_PUBLIC</code>	<code>0x0001</code>	Declared <code>public</code> ; may be accessed from outside its package.
<code>ACC_FINAL</code>	<code>0x0010</code>	Declared <code>final</code> ; no subclasses allowed.
<code>ACC_SUPER</code>	<code>0x0020</code>	Treat superclass methods specially when invoked by the <i>invokespecial</i> instruction.
<code>ACC_INTERFACE</code>	<code>0x0200</code>	Is an interface, not a class.
<code>ACC_ABSTRACT</code>	<code>0x0400</code>	Declared <code>abstract</code> ; may not be instantiated.

An interface is distinguished by its `ACC_INTERFACE` flag being set. If its `ACC_INTERFACE` flag is not set, this `class` file defines a class, not an interface.

If the `ACC_INTERFACE` flag of this `class` file is set, its `ACC_ABSTRACT` flag must also be set ([§2.13.1](#)) and its `ACC_PUBLIC` flag may be set. Such a `class` file may not have any of the other flags in [Table 4.1](#) set.

If the `ACC_INTERFACE` flag of this `class` file is not set, it may have any of the other flags in [Table 4.1](#) set. However, such a `class` file cannot have both its `ACC_FINAL` and `ACC_ABSTRACT` flags set ([§2.8.2](#)).

The setting of the `ACC_SUPER` flag indicates which of two alternative semantics for its *invokespecial* instruction the Java virtual machine is to express; the `ACC_SUPER` flag exists for backward compatibility for code compiled by Sun's older compilers for the Java programming language. All new implementations of the Java virtual machine should implement the semantics for *invokespecial* documented in this specification. All new compilers to the instruction set of the Java virtual machine should set the `ACC_SUPER` flag. Sun's older compilers generated `ClassFile` flags with

`ACC_SUPER` unset. Sun's older Java virtual machine implementations ignore the flag if it is set.

All bits of the `access_flags` item not assigned in [Table 4.1](#) are reserved for future use. They should be set to zero in generated `class` files and should be ignored by Java virtual machine implementations.

`this_class`

The value of the `this_class` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Class_info` ([§4.4.1](#)) structure representing the class or interface defined by this `class` file.

`super_class`

For a class, the value of the `super_class` item either must be zero or must be a valid index into the `constant_pool` table. If the value of the `super_class` item is nonzero, the `constant_pool` entry at that index must be a `CONSTANT_Class_info` ([§4.4.1](#)) structure representing the direct superclass of the class defined by this `class` file. Neither the direct superclass nor any of its superclasses may be a `final` class.

If the value of the `super_class` item is zero, then this `class` file must represent the class `Object`, the only class or interface without a direct superclass.

For an interface, the value of the `super_class` item must always be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Class_info` structure representing the class `Object`.

`interfaces_count`

The value of the `interfaces_count` item gives the number of direct superinterfaces of this class or interface type.

`interfaces[]`

Each value in the `interfaces` array must be a valid index into the `constant_pool` table. The `constant_pool` entry at each value of `interfaces[i]`, where $0 \leq i < \text{interfaces\_count}$, must be a `CONSTANT_Class_info` ([§4.4.1](#)) structure representing an interface that is a direct superinterface of this class or interface type, in the left-to-right order given in the source for the type.

`fields_count`

The value of the `fields_count` item gives the number of `field_info` structures in the `fields` table. The `field_info` ([§4.5](#)) structures represent all fields, both class variables and instance variables, declared by this class or interface type.

`fields[]`

Each value in the `fields` table must be a `field_info` ([§4.5](#)) structure giving a complete description of a field in this class or interface. The `fields` table includes only those fields

that are declared by this class or interface. It does not include items representing fields that are inherited from superclasses or superinterfaces.

`methods_count`

The value of the `methods_count` item gives the number of `method_info` structures in the `methods` table.

`methods[]`

Each value in the `methods` table must be a `method_info` ([§4.6](#)) structure giving a complete description of a method in this class or interface. If the method is not `native` or `abstract`, the Java virtual machine instructions implementing the method are also supplied.

The `method_info` structures represent all methods declared by this class or interface type, including instance methods, class (`static`) methods, instance initialization methods ([§3.9](#)), and any class or interface initialization method ([§3.9](#)). The `methods` table does not include items representing methods that are inherited from superclasses or superinterfaces.

`attributes_count`

The value of the `attributes_count` item gives the number of attributes ([§4.7](#)) in the `attributes` table of this class.

`attributes[]`

Each value of the `attributes` table must be an attribute structure ([§4.7](#)).

The only attributes defined by this specification as appearing in the `attributes` table of a `ClassFile` structure are the `SourceFile` attribute ([§4.7.7](#)) and the `Deprecated` ([§4.7.10](#)) attribute.

A Java virtual machine implementation is required to silently ignore any or all attributes in the `attributes` table of a `ClassFile` structure that it does not recognize. Attributes not defined in this specification are not allowed to affect the semantics of the `class` file, but only to provide additional descriptive information ([§4.7.1](#)).

4.2 The Internal Form of Fully Qualified Class and Interface Names

Class and interface names that appear in `class` file structures are always represented in a fully qualified form ([§2.7.5](#)). Such names are always represented as `CONSTANT_Utf8_info` ([§4.4.7](#)) structures and thus may be drawn, where not further constrained, from the entire Unicode character set. Class names and interfaces are referenced both from those `CONSTANT_NameAndType_info` ([§4.4.6](#)) structures that have such names as part of their descriptor ([§4.3](#)) and from all

`CONSTANT_Class_info` (§4.4.1) structures.

For historical reasons the syntax of fully qualified class and interface names that appear in `class` file structures differs from the familiar syntax of fully qualified names documented in §2.7.5. In this internal form, the ASCII periods (`'.'`) that normally separate the identifiers that make up the fully qualified name are replaced by ASCII forward slashes (`'/'`). For example, the normal fully qualified name of class `Thread` is `java.lang.Thread`. In the form used in descriptors in the `class` file format, a reference to the name of class `Thread` is implemented using a `CONSTANT_Utf8_info` structure representing the string `"java/lang/Thread"`.

4.3 Descriptors

A *descriptor* is a string representing the type of a field or method. Descriptors are represented in the `class` file format using UTF-8 strings (§4.4.7) and thus may be drawn, where not further constrained, from the entire Unicode character set.

4.3.1 Grammar Notation

Descriptors are specified using a grammar. This grammar is a set of productions that describe how sequences of characters can form syntactically correct descriptors of various types. Terminal symbols of the grammar are shown in bold fixed-width font. Nonterminal symbols are shown in *italic* type. The definition of a nonterminal is introduced by the name of the nonterminal being defined, followed by a colon. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. For example, the production:

FieldType:

BaseType

ObjectType

ArrayType

states that a *FieldType* may represent either a *BaseType*, an *ObjectType*, or an *ArrayType*.

A nonterminal symbol on the right-hand side of a production that is followed by an asterisk (*) represents zero or more possibly different values produced from that nonterminal, appended without any intervening space. The production:

MethodDescriptor:

(*ParameterDescriptor**) *ReturnDescriptor*

states that a *MethodDescriptor* represents a left parenthesis, followed by zero or more *ParameterDescriptor* values, followed by a right parenthesis, followed by a *ReturnDescriptor*.

4.3.2 Field Descriptors

A *field descriptor* represents the type of a class, instance, or local variable. It is a series of characters generated by the grammar:

FieldDescriptor:

FieldType

ComponentType:

FieldType

FieldType:

BaseType

ObjectType

ArrayType

BaseType:

B

C

D

F

I

J

S

Z

ObjectType:

L <classname> ;

ArrayType:

[*ComponentType*

The characters of *BaseType*, the L and ; of *ObjectType*, and the [of *ArrayType* are all ASCII characters. The <classname> represents a fully qualified class or interface name. For historical reasons it is encoded in internal form ([§4.2](#)).

The interpretation of the field types is as shown in [Table 4.2](#).

<i>BaseType</i> Character	Type	Interpretation
B	byte	signed byte
C	char	Unicode character
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
L<classname>;	reference	an instance of class <classname>
S	short	signed short
Z	boolean	true or false
[	reference	one array dimension

For example, the descriptor of an instance variable of type `int` is simply `I`. The descriptor of an instance variable of type `Object` is `Ljava/lang/Object;`. Note that the internal form of the fully qualified name for class `Object` is used. The descriptor of an instance variable that is a multidimensional `double` array,

```
double d[][][];
```

is

```
[[[D
```

4.3.3 Method Descriptors

A *method descriptor* represents the parameters that the method takes and the value that it returns:

MethodDescriptor:

(*ParameterDescriptor**) *ReturnDescriptor*

A *parameter descriptor* represents a parameter passed to a method:

ParameterDescriptor:
FieldType

A *return descriptor* represents the type of the value returned from a method. It is a series of characters generated by the grammar:

ReturnDescriptor:
FieldType
V

The character V indicates that the method returns no value (its return type is `void`).

A method descriptor is valid only if it represents method parameters with a total length of 255 or less, where that length includes the contribution for `this` in the case of instance or interface method invocations. The total length is calculated by summing the contributions of the individual parameters, where a parameter of type `long` or `double` contributes two units to the length and a parameter of any other type contributes one unit.

For example, the method descriptor for the method

```
Object mymethod(int i, double d, Thread t)
```

is

```
(IDLjava/lang/Thread;)Ljava/lang/Object;
```

Note that internal forms of the fully qualified names of `Thread` and `Object` are used in the method descriptor.

The method descriptor for `mymethod` is the same whether `mymethod` is a class or an instance method. Although an instance method is passed `this`, a reference to the current class instance, in addition to its intended parameters, that fact is not reflected in the method descriptor. (A reference to `this` is not passed to a class method.) The reference to `this` is passed implicitly by the method invocation instructions of the Java virtual machine used to invoke instance methods.

4.4 The Constant Pool

Java virtual machine instructions do not rely on the runtime layout of classes, interfaces, class

instances, or arrays. Instead, instructions refer to symbolic information in the `constant_pool` table.

All `constant_pool` table entries have the following general format:

```
cp_info {
    u1 tag;
    u1 info[];
}
```

Each item in the `constant_pool` table must begin with a 1-byte tag indicating the kind of `cp_info` entry. The contents of the `info` array vary with the value of `tag`. The valid tags and their values are listed in [Table 4.3](#). Each tag byte must be followed by two or more bytes giving information about the specific constant. The format of the additional information varies with the tag value.

<i>Constant Type</i>	<i>Value</i>
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

4.4.1 The `CONSTANT_Class_info` Structure

The `CONSTANT_Class_info` structure is used to represent a class or an interface:

```
CONSTANT_Class_info {
    u1 tag;
    u2 name_index;
}
```

The items of the `CONSTANT_Class_info` structure are the following:

tag

The **tag** item has the value **CONSTANT_Class** (7).

name_index

The value of the **name_index** item must be a valid index into the **constant_pool** table. The **constant_pool** entry at that index must be a **CONSTANT_Utf8_info** (§4.4.7) structure representing a valid fully qualified class or interface name (§2.8.1) encoded in internal form (§4.2).

Because arrays are objects, the opcodes *anewarray* and *multianewarray* can reference array "classes" via **CONSTANT_Class_info** (§4.4.1) structures in the **constant_pool** table. For such array classes, the name of the class is the descriptor of the array type. For example, the class name representing a two-dimensional **int** array type

```
int[][]
```

is

```
[[I
```

The class name representing the type array of class **Thread**

```
Thread[]
```

is

```
[Ljava/lang/Thread;
```

An array type descriptor is valid only if it represents 255 or fewer dimensions.

4.4.2 The **CONSTANT_Fieldref_info**, **CONSTANT_Methodref_info**, and **CONSTANT_InterfaceMethodref_info** Structures

Fields, methods, and interface methods are represented by similar structures:

```
CONSTANT_Fieldref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
CONSTANT_Methodref_info {
    u1 tag;
    u2 class_index;
```

```

    u2 name_and_type_index;
}
CONSTANT_InterfaceMethodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}

```

The items of these structures are as follows:

tag

The `tag` item of a `CONSTANT_Fieldref_info` structure has the value `CONSTANT_Fieldref` (9).

The `tag` item of a `CONSTANT_Methodref_info` structure has the value `CONSTANT_Methodref` (10).

The `tag` item of a `CONSTANT_InterfaceMethodref_info` structure has the value `CONSTANT_InterfaceMethodref` (11).

class_index

The value of the `class_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Class_info` (§4.4.1) structure representing the class or interface type that contains the declaration of the field or method.

The `class_index` item of a `CONSTANT_Methodref_info` structure must be a class type, not an interface type. The `class_index` item of a `CONSTANT_InterfaceMethodref_info` structure must be an interface type. The `class_index` item of a `CONSTANT_Fieldref_info` structure may be either a class type or an interface type.

name_and_type_index

The value of the `name_and_type_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_NameAndType_info` (§4.4.6) structure. This `constant_pool` entry indicates the name and descriptor of the field or method. In a `CONSTANT_Fieldref_info` the indicated descriptor must be a field descriptor (§4.3.2). Otherwise, the indicated descriptor must be a method descriptor (§4.3.3).

If the name of the method of a `CONSTANT_Methodref_info` structure begins with a '`<`' ('`\u003C`'), then the name must be the special name `<init>`, representing an instance initialization method (§3.9). Such a method must return no value.

4.4.3 The `CONSTANT_String_info` Structure

The `CONSTANT_String_info` structure is used to represent constant objects of the type `String`:

```
CONSTANT_String_info {
    u1 tag;
    u2 string_index;
}
```

The items of the `CONSTANT_String_info` structure are as follows:

tag

The `tag` item of the `CONSTANT_String_info` structure has the value `CONSTANT_String` (8).

string_index

The value of the `string_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.4.7) structure representing the sequence of characters to which the `String` object is to be initialized.

4.4.4 The `CONSTANT_Integer_info` and `CONSTANT_Float_info` Structures

The `CONSTANT_Integer_info` and `CONSTANT_Float_info` structures represent 4-byte numeric (`int` and `float`) constants:

```
CONSTANT_Integer_info {
    u1 tag;
    u4 bytes;
}
CONSTANT_Float_info {
    u1 tag;
    u4 bytes;
}
```

The items of these structures are as follows:

tag

The `tag` item of the `CONSTANT_Integer_info` structure has the value `CONSTANT_Integer` (3).

The `tag` item of the `CONSTANT_Float_info` structure has the value `CONSTANT_Float` (4).

bytes

The `bytes` item of the `CONSTANT_Integer_info` structure represents the value of the `int` constant. The bytes of the value are stored in big-endian (high byte first) order.

The `bytes` item of the `CONSTANT_Float_info` structure represents the value of the `float` constant in IEEE 754 floating-point single format ([§3.3.2](#)). The bytes of the single format representation are stored in big-endian (high byte first) order.

The value represented by the `CONSTANT_Float_info` structure is determined as follows. The bytes of the value are first converted into an `int` constant bits. Then:

- If bits is `0x7f800000`, the `float` value will be positive infinity.
- If bits is `0xff800000`, the `float` value will be negative infinity.
- If bits is in the range `0x7f800001` through `0x7fffffff` or in the range `0xff800001` through `0xffffffff`, the `float` value will be NaN.
- In all other cases, let *s*, *e*, and *m* be three values that might be computed from *bits*:

```
int s = ((bits >> 31) == 0) ? 1 : -1;
int e = ((bits >> 23) & 0xff);
int m = (e == 0) ?
        (bits & 0x7fffffff) << 1 :
        (bits & 0x7fffffff) | 0x800000;
```

Then the `float` value equals the result of the mathematical expression $s \cdot m \cdot 2^{e-150}$.

4.4.5 The `CONSTANT_Long_info` and `CONSTANT_Double_info` Structures

The `CONSTANT_Long_info` and `CONSTANT_Double_info` represent 8-byte numeric (long and double) constants:

```
CONSTANT_Long_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}
CONSTANT_Double_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}
```

All 8-byte constants take up two entries in the `constant_pool` table of the `class` file. If a

`CONSTANT_Long_info` or `CONSTANT_Double_info` structure is the item in the `constant_pool` table at index `n`, then the next usable item in the pool is located at index `n + 2`. The `constant_pool` index `n + 1` must be valid but is considered unusable.²

The items of these structures are as follows:

tag

The tag item of the `CONSTANT_Long_info` structure has the value `CONSTANT_Long` (5).

The tag item of the `CONSTANT_Double_info` structure has the value `CONSTANT_Double` (6).

high_bytes, low_bytes

The unsigned `high_bytes` and `low_bytes` items of the `CONSTANT_Long_info` structure together represent the value of the long constant $((\text{long}) \text{high\_bytes} \ll 32) + \text{low\_bytes}$, where the bytes of each of `high_bytes` and `low_bytes` are stored in big-endian (high byte first) order.

The `high_bytes` and `low_bytes` items of the `CONSTANT_Double_info` structure together represent the double value in IEEE 754 floating-point double format (§3.3.2). The bytes of each item are stored in big-endian (high byte first) order.

The value represented by the `CONSTANT_Double_info` structure is determined as follows. The `high_bytes` and `low_bytes` items are first converted into the long constant bits, which is equal to $((\text{long}) \text{high\_bytes} \ll 32) + \text{low\_bytes}$. Then:

- If bits is `0x7ff0000000000000L`, the double value will be positive infinity.
- If bits is `0xfff0000000000000L`, the double value will be negative infinity.
- If bits is in the range `0x7ff0000000000001L` through `0x7fffffffffffffffffL` or in the range `0xfff0000000000001L` through `0xffffffffffffffffL`, the double value will be NaN.
- In all other cases, let `s`, `e`, and `m` be three values that might be computed from bits:

```
int s = ((bits >> 63) == 0) ? 1 : -1;
int e = (int)((bits >> 52) & 0x7ffL);
long m = (e == 0) ?
    (bits & 0xffffffffffffffffL) << 1 :
    (bits & 0xffffffffffffffffL) |
0x1000000000000000L;
```

Then the floating-point value equals the double value of the mathematical expression $s \cdot m \cdot 2^{e-1075}$.

4.4.6 The `CONSTANT_NameAndType_info` Structure

The `CONSTANT_NameAndType_info` structure is used to represent a field or method, without indicating which class or interface type it belongs to:

```
CONSTANT_NameAndType_info {
    u1 tag;
    u2 name_index;
    u2 descriptor_index;
}
```

The items of the `CONSTANT_NameAndType_info` structure are as follows:

tag

The `tag` item of the `CONSTANT_NameAndType_info` structure has the value `CONSTANT_NameAndType` (12).

name_index

The value of the `name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.4.7) structure representing either a valid field or method name (§2.7) stored as a simple name (§2.7.1), that is, as a Java programming language identifier (§2.2) or as the special method name `<init>` (§3.9).

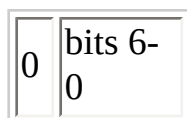
descriptor_index

The value of the `descriptor_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.4.7) structure representing a valid field descriptor (§4.3.2) or method descriptor (§4.3.3).

4.4.7 The `CONSTANT_Utf8_info` Structure

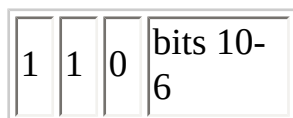
The `CONSTANT_Utf8_info` structure is used to represent constant string values.

UTF-8 strings are encoded so that character sequences that contain only non-null ASCII characters can be represented using only 1 byte per character, but characters of up to 16 bits can be represented. All characters in the range `'\u0001'` to `'\u007F'` are represented by a single byte:

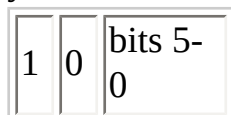


The 7 bits of data in the byte give the value of the character represented. The null character ('`\u0000`') and characters in the range '`\u0080`' to '`\u07FF`' are represented by a pair of bytes `x` and `y`:

`x`:



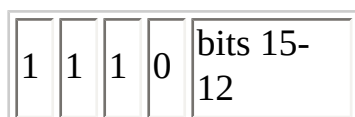
`y`:



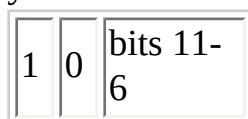
The bytes represent the character with the value $((x \& 0x1f) \ll 6) + (y \& 0x3f)$.

Characters in the range '`\u0800`' to '`\uFFFF`' are represented by 3 bytes `x`, `y`, and `z`:

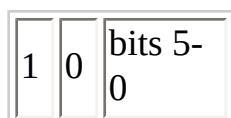
`x`:



`y`:



`z`:



The character with the value $((x \& 0xf) \ll 12) + ((y \& 0x3f) \ll 6) + (z \& 0x3f)$ is represented by the bytes.

The bytes of multibyte characters are stored in the `class` file in big-endian (high byte first) order.

There are two differences between this format and the "standard" UTF-8 format. First, the null byte (`byte`)`0` is encoded using the 2-byte format rather than the 1-byte format, so that Java virtual machine UTF-8 strings never have embedded nulls. Second, only the 1-byte, 2-byte, and 3-byte formats are used. The Java virtual machine does not recognize the longer UTF-8 formats.

For more information regarding the UTF-8 format, see *File System Safe UCS Transformation Format (FSS_UTF)*, X/Open Preliminary Specification (X/Open Company Ltd., Document Number: P316). This information also appears in ISO/IEC 10646, Annex P.

The `CONSTANT_Utf8_info` structure is

```
CONSTANT_Utf8_info {
    u1 tag;
    u2 length;
    u1 bytes[length];
}
```

The items of the `CONSTANT_Utf8_info` structure are the following:

tag

The `tag` item of the `CONSTANT_Utf8_info` structure has the value `CONSTANT_Utf8(1)`.

length

The value of the `length` item gives the number of bytes in the `bytes` array (not the length of the resulting string). The strings in the `CONSTANT_Utf8_info` structure are not null-terminated.

bytes[]

The `bytes` array contains the bytes of the string. No byte may have the value (byte) 0 or lie in the range (byte) 0xf0-(byte) 0xff.

4.5 Fields

Each field is described by a `field_info` structure. No two fields in one `class` file may have the same name and descriptor ([§4.3.2](#)). The format of this structure is

```
field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

The items of the `field_info` structure are as follows:

access_flags

The value of the `access_flags` item is a mask of flags used to denote access permission to and properties of this field. The interpretation of each flag, when set, is

as shown in [Table 4.4](#).

Fields of classes may set any of the flags in [Table 4.4](#). However, a specific field of a class may have at most one of its `ACC_PRIVATE`, `ACC_PROTECTED`, and `ACC_PUBLIC` flags set ([§2.7.4](#)) and may not have both its `ACC_FINAL` and `ACC_VOLATILE` flags set ([§2.9.1](#)).

Flag Name	Value	Interpretation
<code>ACC_PUBLIC</code>	<code>0x0001</code>	Declared <code>public</code> ; may be accessed from outside its package.
<code>ACC_PRIVATE</code>	<code>0x0002</code>	Declared <code>private</code> ; usable only within the defining class.
<code>ACC_PROTECTED</code>	<code>0x0004</code>	Declared <code>protected</code> ; may be accessed within subclasses.
<code>ACC_STATIC</code>	<code>0x0008</code>	Declared <code>static</code> .
<code>ACC_FINAL</code>	<code>0x0010</code>	Declared <code>final</code> ; no further assignment after initialization.
<code>ACC_VOLATILE</code>	<code>0x0040</code>	Declared <code>volatile</code> ; cannot be cached.
<code>ACC_TRANSIENT</code>	<code>0x0080</code>	Declared <code>transient</code> ; not written or read by a persistent object manager.

Fields of classes may set any of the flags in [Table 4.4](#). However, a specific field of a class may have at most one of its `ACC_PRIVATE`, `ACC_PROTECTED`, and `ACC_PUBLIC` flags set ([§2.7.4](#)) and may not have both its `ACC_FINAL` and `ACC_VOLATILE` flags set ([§2.9.1](#)).

All fields of interfaces must have their `ACC_PUBLIC`, `ACC_STATIC`, and `ACC_FINAL` flags set and may not have any of the other flags in [Table 4.4](#) set ([§2.13.3.1](#)).

All bits of the `access_flags` item not assigned in [Table 4.4](#) are reserved for future use. They should be set to zero in generated `class` files and should be ignored by Java virtual machine implementations.

`name_index`

The value of the `name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` ([§4.4.7](#)) structure which must represent a valid field name ([§2.7](#)) stored as a simple name ([§2.7.1](#)), that is, as a Java programming language identifier ([§2.2](#)).

descriptor_index

The value of the `descriptor_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.4.7) structure that must represent a valid field descriptor (§4.3.2).

attributes_count

The value of the `attributes_count` item indicates the number of additional attributes (§4.7) of this field.

attributes[]

Each value of the `attributes` table must be an attribute structure (§4.7). A field can have any number of attributes associated with it.

The attributes defined by this specification as appearing in the `attributes` table of a `field_info` structure are the `ConstantValue` (§4.7.2), `Synthetic` (§4.7.6), and `Deprecated` (§4.7.10) attributes.

A Java virtual machine implementation must recognize and correctly read `ConstantValue` (§4.7.2) attributes found in the `attributes` table of a `field_info` structure. A Java virtual machine implementation is required to silently ignore any or all other attributes in the `attributes` table that it does not recognize. Attributes not defined in this specification are not allowed to affect the semantics of the `class` file, but only to provide additional descriptive information (§4.7.1).

4.6 Methods

Each method, including each instance initialization method (§3.9) and the class or interface initialization method (§3.9), is described by a `method_info` structure. No two methods in one `class` file may have the same name and descriptor (§4.3.3).

The structure has the following format:

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

The items of the `method_info` structure are as follows:

`access_flags`

The value of the `access_flags` item is a mask of flags used to denote access permission to and properties of this method. The interpretation of each flag, when set, is as shown in [Table 4.5](#).

Flag Name	Value	Interpretation
<code>ACC_PUBLIC</code>	<code>0x0001</code>	Declared <code>public</code> ; may be accessed from outside its package.
<code>ACC_PRIVATE</code>	<code>0x0002</code>	Declared <code>private</code> ; accessible only within the defining class.
<code>ACC_PROTECTED</code>	<code>0x0004</code>	Declared <code>protected</code> ; may be accessed within subclasses.
<code>ACC_STATIC</code>	<code>0x0008</code>	Declared <code>static</code> .
<code>ACC_FINAL</code>	<code>0x0010</code>	Declared <code>final</code> ; may not be overridden.
<code>ACC_SYNCHRONIZED</code>	<code>0x0020</code>	Declared <code>synchronized</code> ; invocation is wrapped in a monitor lock.
<code>ACC_NATIVE</code>	<code>0x0100</code>	Declared <code>native</code> ; implemented in a language other than Java.
<code>ACC_ABSTRACT</code>	<code>0x0400</code>	Declared <code>abstract</code> ; no implementation is provided.
<code>ACC_STRICT</code>	<code>0x0800</code>	Declared <code>strictfp</code> ; floating-point mode is FP-strict

Methods of classes may set any of the flags in [Table 4.5](#). However, a specific method of a class may have at most one of its `ACC_PRIVATE`, `ACC_PROTECTED`, and `ACC_PUBLIC` flags set ([§2.7.4](#)). If such a method has its `ACC_ABSTRACT` flag set it may not have any of its `ACC_FINAL`, `ACC_NATIVE`, `ACC_PRIVATE`, `ACC_STATIC`, `ACC_STRICT`, or `ACC_SYNCHRONIZED` flags set ([§2.13.3.2](#)).

All interface methods must have their `ACC_ABSTRACT` and `ACC_PUBLIC` flags set and may not have any of the other flags in [Table 4.5](#) set ([§2.13.3.2](#)).

A specific instance initialization method ([§3.9](#)) may have at most one of its `ACC_PRIVATE`, `ACC_PROTECTED`, and `ACC_PUBLIC` flags set and may also have its `ACC_STRICT` flag set, but may not have any of the other flags in [Table 4.5](#) set.

Class and interface initialization methods ([§3.9](#)) are called implicitly by the Java virtual machine; the value of their `access_flags` item is ignored except for the settings of

the `ACC_STRICT` flag.

All bits of the `access_flags` item not assigned in [Table 4.5](#) are reserved for future use. They should be set to zero in generated `class` files and should be ignored by Java virtual machine implementations.

`name_index`

The value of the `name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` ([§4.4.7](#)) structure representing either one of the special method names ([§3.9](#)), `<init>` or `<clinit>`, or a valid method name in the Java programming language ([§2.7](#)), stored as a simple name ([§2.7.1](#)).

`descriptor_index`

The value of the `descriptor_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` ([§4.4.7](#)) structure representing a valid method descriptor ([§4.3.3](#)).

`attributes_count`

The value of the `attributes_count` item indicates the number of additional attributes ([§4.7](#)) of this method.

`attributes[]`

Each value of the `attributes` table must be an attribute structure ([§4.7](#)). A method can have any number of optional attributes associated with it.

The only attributes defined by this specification as appearing in the `attributes` table of a `method_info` structure are the `Code` ([§4.7.3](#)), `Exceptions` ([§4.7.4](#)), `Synthetic` ([§4.7.6](#)), and `Deprecated` ([§4.7.10](#)) attributes.

A Java virtual machine implementation must recognize and correctly read `Code` ([§4.7.3](#)) and `Exceptions` ([§4.7.4](#)) attributes found in the `attributes` table of a `method_info` structure. A Java virtual machine implementation is required to silently ignore any or all other attributes in the `attributes` table of a `method_info` structure that it does not recognize. Attributes not defined in this specification are not allowed to affect the semantics of the `class` file, but only to provide additional descriptive information ([§4.7.1](#)).

4.7 Attributes

Attributes are used in the `ClassFile` (§4.1), `field_info` (§4.5), `method_info` (§4.6), and `Code_attribute` (§4.7.3) structures of the `class` file format. All attributes have the following general format:

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

For all attributes, the `attribute_name_index` must be a valid unsigned 16-bit index into the constant pool of the class. The `constant_pool` entry at `attribute_name_index` must be a `CONSTANT_Utf8_info` (§4.4.7) structure representing the name of the attribute. The value of the `attribute_length` item indicates the length of the subsequent information in bytes. The length does not include the initial six bytes that contain the `attribute_name_index` and `attribute_length` items.

Certain attributes are predefined as part of the `class` file specification. The predefined attributes are the `SourceFile` (§4.7.7), `ConstantValue` (§4.7.2), `Code` (§4.7.3), `Exceptions` (§4.7.4), `InnerClasses` (§4.7.5), `Synthetic` (§4.7.6), `LineNumberTable` (§4.7.8), `LocalVariableTable` (§4.7.9), and `Deprecated` (§4.7.10) attributes. Within the context of their use in this specification, that is, in the `attributes` tables of the `class` file structures in which they appear, the names of these predefined attributes are reserved.

Of the predefined attributes, the `Code`, `ConstantValue`, and `Exceptions` attributes must be recognized and correctly read by a `class` file reader for correct interpretation of the `class` file by a Java virtual machine implementation. The `InnerClasses` and `Synthetic` attributes must be recognized and correctly read by a `class` file reader in order to properly implement the Java and Java 2 platform class libraries (§3.12). Use of the remaining predefined attributes is optional; a `class` file reader may use the information they contain, or otherwise must silently ignore those attributes.

4.7.1 Defining and Naming New Attributes

Compilers are permitted to define and emit `class` files containing new attributes in the `attributes` tables of `class` file structures. Java virtual machine implementations are permitted to recognize and use new attributes found in the `attributes` tables of `class` file structures. However, any attribute not defined as part of this Java virtual machine specification must not affect the semantics of class or interface types. Java virtual machine implementations are required to silently ignore attributes they do not recognize.

For instance, defining a new attribute to support vendor-specific debugging is permitted. Because Java virtual machine implementations are required to ignore attributes they do not

recognize, `class` files intended for that particular Java virtual machine implementation will be usable by other implementations even if those implementations cannot make use of the additional debugging information that the `class` files contain.

Java virtual machine implementations are specifically prohibited from throwing an exception or otherwise refusing to use `class` files simply because of the presence of some new attribute. Of course, tools operating on `class` files may not run correctly if given `class` files that do not contain all the attributes they require.

Two attributes that are intended to be distinct, but that happen to use the same attribute name and are of the same length, will conflict on implementations that recognize either attribute. Attributes defined other than by Sun must have names chosen according to the package naming convention defined by *The Java Language Specification*. For instance, a new attribute defined by Netscape might have the name `"com.Netscape.new-attribute"`.³

Sun may define additional attributes in future versions of this `class` file specification.

4.7.2 The ConstantValue Attribute

The `ConstantValue` attribute is a fixed-length attribute used in the `attributes` table of the `field_info` (§4.5) structures. A `ConstantValue` attribute represents the value of a constant field that must be (explicitly or implicitly) `static`; that is, the `ACC_STATIC` bit (Table 4.4) in the `flags` item of the `field_info` structure must be set. There can be no more than one `ConstantValue` attribute in the `attributes` table of a given `field_info` structure. The constant field represented by the `field_info` structure is assigned the value referenced by its `ConstantValue` attribute as part of the initialization of the class or interface declaring the constant field (§2.17.4). This occurs immediately prior to the invocation of the class or interface initialization method (§3.9) of that class or interface.

If a `field_info` structure representing a non-`static` field has a `ConstantValue` attribute, then that attribute must silently be ignored. Every Java virtual machine implementation must recognize `ConstantValue` attributes.

The `ConstantValue` attribute has the following format:

```
ConstantValue_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 constantvalue_index;
}
```

The items of the `ConstantValue_attribute` structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.4.7) structure representing the string "ConstantValue".

`attribute_length`

The value of the `attribute_length` item of a `ConstantValue_attribute` structure must be 2.

`constantvalue_index`

The value of the `constantvalue_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index gives the constant value represented by this attribute. The `constant_pool` entry must be of a type appropriate to the field, as shown by [Table 4.6](#).

Field Type	Entry Type
long	CONSTANT_Long
float	CONSTANT_Float
double	CONSTANT_Double
int, short, char, byte, boolean	CONSTANT_Integer
String	CONSTANT_String

4.7.3 The Code Attribute

The **Code** attribute is a variable-length attribute used in the `attributes` table of `method_info` structures. A **Code** attribute contains the Java virtual machine instructions and auxiliary information for a single method, instance initialization method (§3.9), or class or interface initialization method (§3.9). Every Java virtual machine implementation must recognize **Code** attributes. If the method is either **native** or **abstract**, its `method_info` structure must not have a **Code** attribute. Otherwise, its `method_info` structure must have exactly one **Code** attribute.

The **Code** attribute has the following format:

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
```

```

    u2 exception_table_length;
    {
        u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}

```

The items of the `Code_attribute` structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.4.7) structure representing the string "Code".

`attribute_length`

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes.

`max_stack`

The value of the `max_stack` item gives the maximum depth (§3.6.2) of the operand stack of this method at any point during execution of the method.

`max_locals`

The value of the `max_locals` item gives the number of local variables in the local variable array allocated upon invocation of this method, including the local variables used to pass parameters to the method on its invocation.

The greatest local variable index for a value of type `long` or `double` is `max_locals-2`. The greatest local variable index for a value of any other type is `max_locals-1`.

`code_length`

The value of the `code_length` item gives the number of bytes in the `code` array for this method. The value of `code_length` must be greater than zero; the `code` array must not be empty.

`code[]`

The `code` array gives the actual bytes of Java virtual machine code that implement the method.

When the `code` array is read into memory on a byte-addressable machine, if the first

byte of the array is aligned on a 4-byte boundary, the *tableswitch* and *lookupswitch* 32-bit offsets will be 4-byte aligned. (Refer to the descriptions of those instructions for more information on the consequences of `code` array alignment.)

The detailed constraints on the contents of the `code` array are extensive and are given in a separate section ([§4.8](#)).

`exception_table_length`

The value of the `exception_table_length` item gives the number of entries in the `exception_table` table.

`exception_table[]`

Each entry in the `exception_table` array describes one exception handler in the `code` array. The order of the handlers in the `exception_table` array is significant. See [Section 3.10](#) for more details.

Each `exception_table` entry contains the following four items:

`start_pc, end_pc`

The values of the two items `start_pc` and `end_pc` indicate the ranges in the `code` array at which the exception handler is active. The value of `start_pc` must be a valid index into the `code` array of the opcode of an instruction. The value of `end_pc` either must be a valid index into the `code` array of the opcode of an instruction or must be equal to `code_length`, the length of the `code` array. The value of `start_pc` must be less than the value of `end_pc`.

The `start_pc` is inclusive and `end_pc` is exclusive; that is, the exception handler must be active while the program counter is within the interval `[start_pc, end_pc)`.⁴

`handler_pc`

The value of the `handler_pc` item indicates the start of the exception handler. The value of the item must be a valid index into the `code` array and must be the index of the opcode of an instruction.

`catch_type`

If the value of the `catch_type` item is nonzero, it must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Class_info` ([§4.4.1](#)) structure representing a class of exceptions that this exception handler is designated to catch. This class must be the class `Throwable` or one of its subclasses. The exception handler will be called only if the thrown exception is an instance of the given class or one of its subclasses.

If the value of the `catch_type` item is zero, this exception handler is called for all exceptions. This is used to implement `finally` (see [Section 7.13, "Compiling finally"](#)).

`attributes_count`

The value of the `attributes_count` item indicates the number of attributes of the `Code` attribute.

`attributes[]`

Each value of the `attributes` table must be an attribute structure ([§4.7](#)). A `Code` attribute can have any number of optional attributes associated with it.

Currently, the `LineNumberTable` ([§4.7.8](#)) and `LocalVariableTable` ([§4.7.9](#)) attributes, both of which contain debugging information, are defined and used with the `Code` attribute.

A Java virtual machine implementation is permitted to silently ignore any or all attributes in the `attributes` table of a `Code` attribute. Attributes not defined in this specification are not allowed to affect the semantics of the `class` file, but only to provide additional descriptive information ([§4.7.1](#)).

4.7.4 The Exceptions Attribute

The `Exceptions` attribute is a variable-length attribute used in the `attributes` table of a `method_info` ([§4.6](#)) structure. The `Exceptions` attribute indicates which checked exceptions a method may throw. There may be at most one `Exceptions` attribute in each `method_info` structure.

The `Exceptions` attribute has the following format:

```
Exceptions_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_exceptions;
    u2 exception_index_table[number_of_exceptions];
}
```

The items of the `Exceptions_attribute` structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be the `CONSTANT_Utf8_info` ([§4.4.7](#)) structure representing the string "Exceptions".

attribute_length

The value of the `attribute_length` item indicates the attribute length, excluding the initial six bytes.

number_of_exceptions

The value of the `number_of_exceptions` item indicates the number of entries in the `exception_index_table`.

exception_index_table[]

Each value in the `exception_index_table` array must be a valid index into the `constant_pool` table. The `constant_pool` entry referenced by each table item must be a `CONSTANT_Class_info` (§4.4.1) structure representing a class type that this method is declared to throw.

A method should throw an exception only if at least one of the following three criteria is met:

- The exception is an instance of `RuntimeException` or one of its subclasses.
- The exception is an instance of `Error` or one of its subclasses.
- The exception is an instance of one of the exception classes specified in the `exception_index_table` just described, or one of their subclasses.

These requirements are not enforced in the Java virtual machine; they are enforced only at compile time.

4.7.5 The InnerClasses Attribute

The `InnerClasses` attribute⁵ is a variable-length attribute in the `attributes` table of the `ClassFile` (§4.1) structure. If the constant pool of a class or interface refers to any class or interface that is not a member of a package, its `ClassFile` structure must have exactly one `InnerClasses` attribute in its `attributes` table.

The `InnerClasses` attribute has the following format:

```

InnerClasses_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_classes;
    { u2 inner_class_info_index;
      u2 outer_class_info_index;
      u2 inner_name_index;
      u2 inner_class_access_flags;
    } classes[number_of_classes];
}

```

The items of the `InnerClasses_attribute` structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.4.7) structure representing the string "InnerClasses".

`attribute_length`

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes.

`number_of_classes`

The value of the `number_of_classes` item indicates the number of entries in the `classes` array.

`classes[]`

Every `CONSTANT_Class_info` entry in the `constant_pool` table which represents a class or interface `C` that is not a package member must have exactly one corresponding entry in the `classes` array.

If a class has members that are classes or interfaces, its `constant_pool` table (and hence its `InnerClasses` attribute) must refer to each such member, even if that member is not otherwise mentioned by the class. These rules imply that a nested class or interface member will have `InnerClasses` information for each enclosing class and for each immediate member.

Each `classes` array entry contains the following four items:

`inner_class_info_index`

The value of the `inner_class_info_index` item must be zero or a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Class_info` (§4.4.1) structure representing `C`. The remaining items in the `classes` array entry give information about `C`.

`outer_class_info_index`

If `C` is not a member, the value of the `outer_class_info_index` item must be zero. Otherwise, the value of the `outer_class_info_index` item must be a valid index into the `constant_pool` table, and the entry at that index must be a `CONSTANT_Class_info` (§4.4.1) structure representing the class or interface of which `C` is a member.

`inner_name_index`

If `C` is anonymous, the value of the `inner_name_index` item must be zero.

Otherwise, the value of the `inner_name_index` item must be a valid index into the `constant_pool` table, and the entry at that index must be a `CONSTANT_Utf8_info` (§4.4.7) structure that represents the original simple name of `C`, as given in the source code from which this `class` file was compiled.

`inner_class_access_flags`

The value of the `inner_class_access_flags` item is a mask of flags used to denote access permissions to and properties of class or interface `C` as declared in the source code from which this `class` file was compiled. It is used by compilers to recover the original information when source code is not available. The flags are shown in [Table 4.7](#).

Flag Name	Value	Meaning
<code>ACC_PUBLIC</code>	<code>0x0001</code>	Marked or implicitly <code>public</code> in source.
<code>ACC_PRIVATE</code>	<code>0x0002</code>	Marked <code>private</code> in source.
<code>ACC_PROTECTED</code>	<code>0x0004</code>	Marked <code>protected</code> in source.
<code>ACC_STATIC</code>	<code>0x0008</code>	Marked or implicitly <code>static</code> in source.
<code>ACC_FINAL</code>	<code>0x0010</code>	Marked <code>final</code> in source.
<code>ACC_INTERFACE</code>	<code>0x0200</code>	Was an <code>interface</code> in source.
<code>ACC_ABSTRACT</code>	<code>0x0400</code>	Marked or implicitly <code>abstract</code> in source.

All bits of the `inner_class_access_flags` item not assigned in [Table 4.7](#) are reserved for future use. They should be set to zero in generated `class` files and should be ignored by Java virtual machine implementations.

The Java virtual machine does not currently check the consistency of the `InnerClasses` attribute with any `class` file actually representing a class or interface referenced by the attribute.

4.7.6 The Synthetic Attribute

The `Synthetic` attribute⁶ is a fixed-length attribute in the `attributes` table of `ClassFile` (§4.1), `field_info` (§4.5), and `method_info` (§4.6) structures. A class member that does not appear in the source code must be marked using a `Synthetic` attribute.

The `Synthetic` attribute has the following format:

```
Synthetic_attribute {
    u2 attribute_name_index;
```



```

        u4 attribute_length;
    }

```

The items of the `Synthetic_attribute` structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.4.7) structure representing the string "Synthetic".

`attribute_length`

The value of the `attribute_length` item is zero.

4.7.7 The `SourceFile` Attribute

The `SourceFile` attribute is an optional fixed-length attribute in the `attributes` table of the `ClassFile` (§4.1) structure. There can be no more than one `SourceFile` attribute in the `attributes` table of a given `ClassFile` structure.

The `SourceFile` attribute has the following format:

```

SourceFile_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 sourcefile_index;
}

```

The items of the `SourceFile_attribute` structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.4.7) structure representing the string "SourceFile".

`attribute_length`

The value of the `attribute_length` item of a `SourceFile_attribute` structure must be 2.

`sourcefile_index`

The value of the `sourcefile_index` item must be a valid index into the `constant_pool` table. The constant pool entry at that index must be a `CONSTANT_Utf8_info` (§4.4.7) structure representing a string.

The string referenced by the `sourcefile_index` item will be interpreted as

indicating the name of the source file from which this `class` file was compiled. It will not be interpreted as indicating the name of a directory containing the file or an absolute path name for the file; such platform-specific additional information must be supplied by the runtime interpreter or development tool at the time the file name is actually used.

4.7.8 The `LineNumberTable` Attribute

The `LineNumberTable` attribute is an optional variable-length attribute in the `attributes` table of a `Code` ([§4.7.3](#)) attribute. It may be used by debuggers to determine which part of the Java virtual machine `code` array corresponds to a given line number in the original source file. If `LineNumberTable` attributes are present in the `attributes` table of a given `Code` attribute, then they may appear in any order. Furthermore, multiple `LineNumberTable` attributes may together represent a given line of a source file; that is, `LineNumberTable` attributes need not be one-to-one with source lines.

The `LineNumberTable` attribute has the following format:

```
LineNumberTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 line_number_table_length;
    { u2 start_pc;
      u2 line_number;
    } line_number_table[line_number_table_length];
}
```

The items of the `LineNumberTable_attribute` structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` ([§4.4.7](#)) structure representing the string `"LineNumberTable"`.

`attribute_length`

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes.

`line_number_table_length`

The value of the `line_number_table_length` item indicates the number of entries in the `line_number_table` array.

`line_number_table[]`

Each entry in the `line_number_table` array indicates that the line number in the original source file changes at a given point in the `code` array. Each `line_number_table` entry must contain the following two items:

`start_pc`

The value of the `start_pc` item must indicate the index into the `code` array at which the code for a new line in the original source file begins. The value of `start_pc` must be less than the value of the `code_length` item of the `Code` attribute of which this `LineNumberTable` is an attribute.

`line_number`

The value of the `line_number` item must give the corresponding line number in the original source file.

4.7.9 The `LocalVariableTable` Attribute

The `LocalVariableTable` attribute is an optional variable-length attribute of a `Code` ([§4.7.3](#)) attribute. It may be used by debuggers to determine the value of a given local variable during the execution of a method. If `LocalVariableTable` attributes are present in the `attributes` table of a given `Code` attribute, then they may appear in any order. There may be no more than one `LocalVariableTable` attribute per local variable in the `Code` attribute.

The `LocalVariableTable` attribute has the following format:

```
LocalVariableTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 local_variable_table_length;
    {
        u2 start_pc;
        u2 length;
        u2 name_index;
        u2 descriptor_index;
        u2 index;
    } local_variable_table[local_variable_table_length];
}
```

The items of the `LocalVariableTable_attribute` structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` ([§4.4.7](#)) structure representing the string `"LocalVariableTable"`.

attribute_length

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes.

local_variable_table_length

The value of the `local_variable_table_length` item indicates the number of entries in the `local_variable_table` array.

local_variable_table[]

Each entry in the `local_variable_table` array indicates a range of `code` array offsets within which a local variable has a value. It also indicates the index into the local variable array of the current frame at which that local variable can be found. Each entry must contain the following five items:

start_pc, length

The given local variable must have a value at indices into the `code` array in the interval `[start_pc, start_pc+length]`, that is, between `start_pc` and `start_pc+length` inclusive. The value of `start_pc` must be a valid index into the `code` array of this `Code` attribute and must be the index of the opcode of an instruction. Either the value of `start_pc+length` must be a valid index into the `code` array of this `Code` attribute and be the index of the opcode of an instruction, or it must be the first index beyond the end of that `code` array.

name_index, descriptor_index

The value of the `name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must contain a `CONSTANT_Utf8_info` (§4.4.7) structure representing a valid local variable name stored as a simple name (§2.7.1).

The value of the `descriptor_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must contain a `CONSTANT_Utf8_info` (§4.4.7) structure representing a field descriptor (§4.3.2) encoding the type of a local variable in the source program.

index

The given local variable must be at `index` in the local variable array of the current frame. If the local variable at `index` is of type `double` or `long`, it occupies both `index` and `index+1`.

4.7.10 The Deprecated Attribute

The `Deprecated` attribute⁷ is an optional fixed-length attribute in the `attributes` table of `ClassFile` (§4.1), `field_info` (§4.5), and `method_info` (§4.6) structures. A class, interface, method, or field may be marked using a `Deprecated` attribute to indicate that the

class, interface, method, or field has been superseded. A runtime interpreter or tool that reads the `class` file format, such as a compiler, can use this marking to advise the user that a superseded class, interface, method, or field is being referred to. The presence of a `Deprecated` attribute does not alter the semantics of a class or interface.

The `Deprecated` attribute has the following format:

```
Deprecated_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
}
```

The items of the `Deprecated_attribute` structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.4.7) structure representing the string "Deprecated".

`attribute_length`

The value of the `attribute_length` item is zero.

4.8 Constraints on Java Virtual Machine Code

The Java virtual machine code for a method, instance initialization method (§3.9), or class or interface initialization method (§3.9) is stored in the `code` array of the `Code` attribute of a `method_info` structure of a `class` file. This section describes the constraints associated with the contents of the `Code_attribute` structure.

4.8.1 Static Constraints

The *static constraints* on a `class` file are those defining the well-formedness of the file. With the exception of the static constraints on the Java virtual machine code of the `class` file, these constraints have been given in the previous section. The static constraints on the Java virtual machine code in a `class` file specify how Java virtual machine instructions must be laid out in the `code` array and what the operands of individual instructions must be.

The static constraints on the instructions in the `code` array are as follows:

- The `code` array must not be empty, so the `code_length` item cannot have the value 0.

- The value of the `code_length` item must be less than 65536.
- The opcode of the first instruction in the `code` array begins at index 0.
- Only instances of the instructions documented in [Section 6.4](#) may appear in the `code` array. Instances of instructions using the reserved opcodes ([§6.2](#)) or any opcodes not documented in this specification may not appear in the `code` array.
- For each instruction in the `code` array except the last, the index of the opcode of the next instruction equals the index of the opcode of the current instruction plus the length of that instruction, including all its operands. The *wide* instruction is treated like any other instruction for these purposes; the opcode specifying the operation that a *wide* instruction is to modify is treated as one of the operands of that *wide* instruction. That opcode must never be directly reachable by the computation.
- The last byte of the last instruction in the `code` array must be the byte at index `code_length-1`.

The static constraints on the operands of instructions in the `code` array are as follows:

- The target of each jump and branch instruction (*jsr*, *jsr_w*, *goto*, *goto_w*, *ifeq*, *ifne*, *ifle*, *iflt*, *ifge*, *ifgt*, *ifnull*, *ifnonnull*, *if_icmpeq*, *if_icmpne*, *if_icmple*, *if_icmplt*, *if_icmpge*, *if_icmpgt*, *if_acmpeq*, *if_acmpne*) must be the opcode of an instruction within this method. The target of a jump or branch instruction must never be the opcode used to specify the operation to be modified by a *wide* instruction; a jump or branch target may be the *wide* instruction itself.
- Each target, including the default, of each *tableswitch* instruction must be the opcode of an instruction within this method. Each *tableswitch* instruction must have a number of entries in its jump table that is consistent with the value of its *low* and *high* jump table operands, and its *low* value must be less than or equal to its *high* value. No target of a *tableswitch* instruction may be the opcode used to specify the operation to be modified by a *wide* instruction; a *tableswitch* target may be a *wide* instruction itself.
- Each target, including the default, of each *lookupswitch* instruction must be the opcode of an instruction within this method. Each *lookupswitch* instruction must have a number of *match-offset* pairs that is consistent with the value of its *npairs* operand. The *match-offset* pairs must be sorted in increasing numerical order by signed *match* value. No target of a *lookupswitch* instruction may be the opcode used to specify the operation to be modified by a *wide* instruction; a *lookupswitch* target may be a *wide* instruction itself.
- The operand of each *ldc* instruction must be a valid index into the `constant_pool` table. The operands of each *ldc_w* instruction must represent a valid index into the `constant_pool` table. In both cases the constant pool entry referenced by that index must be of type `CONSTANT_Integer`, `CONSTANT_Float`, or

CONSTANT_String.

- The operands of each *ldc2_w* instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_Long` or `CONSTANT_Double`. In addition, the subsequent constant pool index must also be a valid index into the constant pool, and the constant pool entry at that index must not be used.
- The operands of each *getfield*, *putfield*, *getstatic*, and *putstatic* instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_Fieldref`.
- The indexbyte operands of each *invokevirtual*, *invokespecial*, and *invokestatic* instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_Methodref`.
- Only the *invokespecial* instruction is allowed to invoke an instance initialization method ([§3.9](#)). No other method whose name begins with the character '`<`' ('`\u003c`') may be called by the method invocation instructions. In particular, the class or interface initialization method specially named `<clinit>` is never called explicitly from Java virtual machine instructions, but only implicitly by the Java virtual machine itself.
- The indexbyte operands of each *invokeinterface* instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_InterfaceMethodref`. The value of the *count* operand of each *invokeinterface* instruction must reflect the number of local variables necessary to store the arguments to be passed to the interface method, as implied by the descriptor of the `CONSTANT_NameAndType_info` structure referenced by the `CONSTANT_InterfaceMethodref` constant pool entry. The fourth operand byte of each *invokeinterface* instruction must have the value zero.
- The operands of each *instanceof*, *checkcast*, *new*, and *anewarray* instruction and the indexbyte operands of each *multianewarray* instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_Class`.
- No *anewarray* instruction may be used to create an array of more than 255 dimensions.
- No *new* instruction may reference a `CONSTANT_Class` `constant_pool` table entry representing an array class. The *new* instruction cannot be used to create an array. The *new* instruction also cannot be used to create an instance of an interface or an instance of an `abstract` class.
- A *multianewarray* instruction must be used only to create an array of a type that has at

least as many dimensions as the value of its *dimensions* operand. That is, while a *multianewarray* instruction is not required to create all of the dimensions of the array type referenced by its *indexbyte* operands, it must not attempt to create more dimensions than are in the array type. The *dimensions* operand of each *multianewarray* instruction must not be zero.

- The *atype* operand of each *newarray* instruction must take one of the values `T_BOOLEAN` (4), `T_CHAR` (5), `T_FLOAT` (6), `T_DOUBLE` (7), `T_BYTE` (8), `T_SHORT` (9), `T_INT` (10), or `T_LONG` (11).
- The index operand of each *iload*, *fload*, *aload*, *istore*, *fstore*, *astore*, *iinc*, and *ret* instruction must be a nonnegative integer no greater than `max_locals-1`.
- The implicit index of each *iload*_*<n>*, *fload*_*<n>*, *aload*_*<n>*, *istore*_*<n>*, *fstore*_*<n>*, and *astore*_*<n>* instruction must be no greater than the value of `max_locals-1`.
- The index operand of each *lload*, *dload*, *lstore*, and *dstore* instruction must be no greater than the value of `max_locals-2`.
- The implicit index of each *lload*_*<n>*, *dload*_*<n>*, *lstore*_*<n>*, and *dstore*_*<n>* instruction must be no greater than the value of `max_locals-2`.
- The *indexbyte* operands of each *wide* instruction modifying an *iload*, *fload*, *aload*, *istore*, *fstore*, *astore*, *ret*, or *iinc* instruction must represent a nonnegative integer no greater than `max_locals-1`. The *indexbyte* operands of each *wide* instruction modifying an *lload*, *dload*, *lstore*, or *dstore* instruction must represent a nonnegative integer no greater than `max_locals-2`.

4.8.2 Structural Constraints

The structural constraints on the `code` array specify constraints on relationships between Java virtual machine instructions. The structural constraints are as follows:

- Each instruction must only be executed with the appropriate type and number of arguments in the operand stack and local variable array, regardless of the execution path that leads to its invocation. An instruction operating on values of type `int` is also permitted to operate on values of type `boolean`, `byte`, `char`, and `short`. (As noted in [§3.3.4](#) and [§3.11.1](#), the Java virtual machine internally converts values of types `boolean`, `byte`, `char`, and `short` to type `int`.)
- If an instruction can be executed along several different execution paths, the operand stack must have the same depth ([§3.6.2](#)) prior to the execution of the instruction, regardless of the path taken.

- At no point during execution can the order of the local variable pair holding a value of type `long` or `double` be reversed or the pair split up. At no point can the local variables of such a pair be operated on individually.
- No local variable (or local variable pair, in the case of a value of type `long` or `double`) can be accessed before it is assigned a value.
- At no point during execution can the operand stack grow to a depth (§3.6.2) greater than that implied by the `max_stack` item.
- At no point during execution can more values be popped from the operand stack than it contains.
- Each *invokespecial* instruction must name an instance initialization method (§3.9), a method in the current class, or a method in a superclass of the current class.
- When the instance initialization method (§3.9) is invoked, an uninitialized class instance must be in an appropriate position on the operand stack. An instance initialization method must never be invoked on an initialized class instance.
- When any instance method is invoked or when any instance variable is accessed, the class instance that contains the instance method or instance variable must already be initialized.
- There must never be an uninitialized class instance on the operand stack or in a local variable when any backwards branch is taken. There must never be an uninitialized class instance in a local variable in code protected by an exception handler. However, an uninitialized class instance may be on the operand stack in code protected by an exception handler. When an exception is thrown, the contents of the operand stack are discarded.
- Each instance initialization method (§3.9), except for the instance initialization method derived from the constructor of class `Object`, must call either another instance initialization method of `this` or an instance initialization method of its direct superclass `super` before its instance members are accessed. However, instance fields of `this` that are declared in the current class may be assigned before calling any instance initialization method.
- The arguments to each method invocation must be method invocation compatible (§2.6.8) with the method descriptor (§4.3.3).
- The type of every class instance that is the target of a method invocation instruction must be assignment compatible (§2.6.7) with the class or interface type specified in the instruction.

- Each return instruction must match its method's return type. If the method returns a `boolean`, `byte`, `char`, `short`, or `int`, only the *ireturn* instruction may be used. If the method returns a `float`, `long`, or `double`, only an *freturn*, *lreturn*, or *dreturn* instruction, respectively, may be used. If the method returns a `reference` type, it must do so using an *areturn* instruction, and the type of the returned value must be assignment compatible (§2.6.7) with the return descriptor (§4.3.3) of the method. All instance initialization methods, class or interface initialization methods, and methods declared to return `void` must use only the *return* instruction.
- If *getfield* or *putfield* is used to access a `protected` field of a superclass, then the type of the class instance being accessed must be the same as or a subclass of the current class. If *invokevirtual* or *invokespecial* is used to access a `protected` method of a superclass, then the type of the class instance being accessed must be the same as or a subclass of the current class.
- The type of every class instance accessed by a *getfield* instruction or modified by a *putfield* instruction must be assignment compatible (§2.6.7) with the class type specified in the instruction.
- The type of every value stored by a *putfield* or *putstatic* instruction must be compatible with the descriptor of the field (§4.3.2) of the class instance or class being stored into. If the descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the value must be an `int`. If the descriptor type is `float`, `long`, or `double`, then the value must be a `float`, `long`, or `double`, respectively. If the descriptor type is a `reference` type, then the value must be of a type that is assignment compatible (§2.6.7) with the descriptor type.
- The type of every value stored into an array of type `reference` by an *aastore* instruction must be assignment compatible (§2.6.7) with the component type of the array.
- Each *athrow* instruction must throw only values that are instances of class `Throwable` or of subclasses of `Throwable`.
- Execution never falls off the bottom of the `code` array.
- No return address (a value of type `returnAddress`) may be loaded from a local variable.
- The instruction following each *jsr* or *jsr_w* instruction may be returned to only by a single *ret* instruction.
- No *jsr* or *jsr_w* instruction may be used to recursively call a subroutine if that

subroutine is already present in the subroutine call chain. (Subroutines can be nested when using `try-finally` constructs from within a `finally` clause. For more information on Java virtual machine subroutines, see [§4.9.6](#).)

- Each instance of type `returnAddress` can be returned to at most once. If a `ret` instruction returns to a point in the subroutine call chain above the `ret` instruction corresponding to a given instance of type `returnAddress`, then that instance can never be used as a return address.
-

4.9 Verification of `class` Files

Even though Sun's compiler for the Java programming language attempts to produce only class files that satisfy all the static constraints in the previous sections, the Java virtual machine has no guarantee that any file it is asked to load was generated by that compiler or is properly formed. Applications such as Sun's HotJava World Wide Web browser do not download source code, which they then compile; these applications download already-compiled `class` files. The HotJava browser needs to determine whether the `class` file was produced by a trustworthy compiler or by an adversary attempting to exploit the virtual machine.

An additional problem with compile-time checking is version skew. A user may have successfully compiled a class, say `PurchaseStockOptions`, to be a subclass of `TradingClass`. But the definition of `TradingClass` might have changed since the time the class was compiled in a way that is not compatible with preexisting binaries. Methods might have been deleted or had their return types or modifiers changed. Fields might have changed types or changed from instance variables to class variables. The access modifiers of a method or variable may have changed from `public` to `private`. For a discussion of these issues, see Chapter 13, "Binary Compatibility," in the first edition of *The Java Language Specification* or the equivalent chapter in the second edition.

Because of these potential problems, the Java virtual machine needs to verify for itself that the desired constraints are satisfied by the `class` files it attempts to incorporate. A Java virtual machine implementation verifies that each `class` file satisfies the necessary constraints at linking time ([§2.17.3](#)). Structural constraints on the Java virtual machine code may be checked using a simple theorem prover.

Linking-time verification enhances the performance of the interpreter. Expensive checks that would otherwise have to be performed to verify constraints at run time for each interpreted instruction can be eliminated. The Java virtual machine can assume that these checks have already been performed. For example, the Java virtual machine will already know the following:

- There are no operand stack overflows or underflows.
- All local variable uses and stores are valid.
- The arguments to all the Java virtual machine instructions are of valid types.

Sun's `class` file verifier is independent of any compiler. It should certify all code generated by Sun's compiler for the Java programming language; it should also certify code that other compilers can generate, as well as code that the current compiler could not possibly generate. Any `class` file that satisfies the structural criteria and static constraints will be certified by the verifier.

The `class` file verifier is also independent of the Java programming language. Programs written in other languages can be compiled into the `class` file format, but will pass verification only if all the same constraints are satisfied.

4.9.1 The Verification Process

The `class` file verifier operates in four passes:

Pass 1:

When a prospective `class` file is loaded ([§2.17.2](#)) by the Java virtual machine, the Java virtual machine first ensures that the file has the basic format of a `class` file. The first four bytes must contain the right magic number. All recognized attributes must be of the proper length. The `class` file must not be truncated or have extra bytes at the end. The constant pool must not contain any superficially unrecognizable information.

While `class` file verification properly occurs during class linking ([§2.17.3](#)), this check for basic `class` file integrity is necessary for any interpretation of the `class` file contents and can be considered to be logically part of the verification process.

Pass 2:

When the `class` file is linked, the verifier performs all additional verification that can be done without looking at the `code` array of the `Code` attribute ([§4.7.3](#)). The checks performed by this pass include the following:

- Ensuring that `final` classes are not subclassed and that `final` methods are not overridden.
- Checking that every class (except `Object`) has a direct superclass.
- Ensuring that the constant pool satisfies the documented static constraints: for example, that each `CONSTANT_Class_info` structure in the constant pool contains in its `name_index` item a valid constant pool index for a `CONSTANT_Utf8_info` structure.

- Checking that all field references and method references in the constant pool have valid names, valid classes, and a valid type descriptor.

Note that when it looks at field and method references, this pass does not check to make sure that the given field or method actually exists in the given class, nor does it check that the type descriptors given refer to real classes. It checks only that these items are well formed. More detailed checking is delayed until passes 3 and 4.

Pass 3:

During linking, the verifier checks the `code` array of the `Code` attribute for each method of the `class` file by performing data-flow analysis on each method. The verifier ensures that at any given point in the program, no matter what code path is taken to reach that point, the following is true:

- The operand stack is always the same size and contains the same types of values.
- No local variable is accessed unless it is known to contain a value of an appropriate type.
- Methods are invoked with the appropriate arguments.
- Fields are assigned only using values of appropriate types.
- All opcodes have appropriate type arguments on the operand stack and in the local variable array.

For further information on this pass, see [Section 4.9.2, "The Bytecode Verifier."](#)

Pass 4:

For efficiency reasons, certain tests that could in principle be performed in Pass 3 are delayed until the first time the code for the method is actually invoked. In so doing, Pass 3 of the verifier avoids loading `class` files unless it has to.

For example, if a method invokes another method that returns an instance of class `A`, and that instance is assigned only to a field of the same type, the verifier does not bother to check if the class `A` actually exists. However, if it is assigned to a field of the type `B`, the definitions of both `A` and `B` must be loaded in to ensure that `A` is a subclass of `B`.

Pass 4 is a virtual pass whose checking is done by the appropriate Java virtual machine instructions. The first time an instruction that references a type is executed, the executing instruction does the following:

- Loads in the definition of the referenced type if it has not already been loaded.
- Checks that the currently executing type is allowed to reference the type.

The first time an instruction invokes a method, or accesses or modifies a field, the executing

instruction does the following:

- Ensures that the referenced method or field exists in the given class.
- Checks that the referenced method or field has the indicated descriptor.
- Checks that the currently executing method has access to the referenced method or field.

The Java virtual machine does not have to check the type of the object on the operand stack. That check has already been done by Pass 3. Errors that are detected in Pass 4 cause instances of subclasses of `LinkageError` to be thrown.

A Java virtual machine implementation is allowed to perform any or all of the Pass 4 steps as part of Pass 3; see [2.17.1, "Virtual Machine Start-up"](#) for an example and more discussion.

In one of Sun's Java virtual machine implementations, after the verification has been performed, the instruction in the Java virtual machine code is replaced with an alternative form of the instruction. This alternative instruction indicates that the verification needed by this instruction has taken place and does not need to be performed again. Subsequent invocations of the method will thus be faster. It is illegal for these alternative instruction forms to appear in `class` files, and they should never be encountered by the verifier.

4.9.2 The Bytecode Verifier

As indicated earlier, Pass 3 of the verification process is the most complex of the four passes of `class` file verification. This section looks at the verification of Java virtual machine code in Pass 3 in more detail.

The code for each method is verified independently. First, the bytes that make up the code are broken up into a sequence of instructions, and the index into the `code` array of the start of each instruction is placed in an array. The verifier then goes through the code a second time and parses the instructions. During this pass a data structure is built to hold information about each Java virtual machine instruction in the method. The operands, if any, of each instruction are checked to make sure they are valid. For instance:

- Branches must be within the bounds of the `code` array for the method.
- The targets of all control-flow instructions are each the start of an instruction. In the case of a *wide* instruction, the *wide* opcode is considered the start of the instruction, and the opcode giving the operation modified by that *wide* instruction is not considered to start an instruction. Branches into the middle of an instruction are disallowed.
- No instruction can access or modify a local variable at an index greater than or equal to the number of local variables that its method indicates it allocates.

- All references to the constant pool must be to an entry of the appropriate type. For example: the instruction *ldc* can be used only for data of type `int` or `float` or for instances of class `String`; the instruction *getfield* must reference a field.
- The code does not end in the middle of an instruction.
- Execution cannot fall off the end of the code.
- For each exception handler, the starting and ending point of code protected by the handler must be at the beginning of an instruction or, in the case of the ending point, immediately past the end of the code. The starting point must be before the ending point. The exception handler code must start at a valid instruction, and it may not start at an opcode being modified by the *wide* instruction.

For each instruction of the method, the verifier records the contents of the operand stack and the contents of the local variable array prior to the execution of that instruction. For the operand stack, it needs to know the stack height and the type of each value on it. For each local variable, it needs to know either the type of the contents of that local variable or that the local variable contains an unusable or unknown value (it might be uninitialized). The bytecode verifier does not need to distinguish between the integral types (e.g., `byte`, `short`, `char`) when determining the value types on the operand stack.

Next, a data-flow analyzer is initialized. For the first instruction of the method, the local variables that represent parameters initially contain values of the types indicated by the method's type descriptor; the operand stack is empty. All other local variables contain an illegal value. For the other instructions, which have not been examined yet, no information is available regarding the operand stack or local variables.

Finally, the data-flow analyzer is run. For each instruction, a "changed" bit indicates whether this instruction needs to be looked at. Initially, the "changed" bit is set only for the first instruction. The data-flow analyzer executes the following loop:

1. Select a virtual machine instruction whose "changed" bit is set. If no instruction remains whose "changed" bit is set, the method has successfully been verified. Otherwise, turn off the "changed" bit of the selected instruction.
2. Model the effect of the instruction on the operand stack and local variable array by doing the following:
 - If the instruction uses values from the operand stack, ensure that there are a sufficient number of values on the stack and that the top values on the stack are of an appropriate type. Otherwise, verification fails.
 - If the instruction uses a local variable, ensure that the specified local variable contains a value of the appropriate type. Otherwise, verification fails.

- If the instruction pushes values onto the operand stack, ensure that there is sufficient room on the operand stack for the new values. Add the indicated types to the top of the modeled operand stack.
 - If the instruction modifies a local variable, record that the local variable now contains the new type.
3. Determine the instructions that can follow the current instruction. Successor instructions can be one of the following:
- The next instruction, if the current instruction is not an unconditional control transfer instruction (for instance *goto*, *return*, or *athrow*). Verification fails if it is possible to "fall off" the last instruction of the method.
 - The target(s) of a conditional or unconditional branch or switch.
 - Any exception handlers for this instruction.
4. Merge the state of the operand stack and local variable array at the end of the execution of the current instruction into each of the successor instructions. In the special case of control transfer to an exception handler, the operand stack is set to contain a single object of the exception type indicated by the exception handler information.
- If this is the first time the successor instruction has been visited, record that the operand stack and local variable values calculated in steps 2 and 3 are the state of the operand stack and local variable array prior to executing the successor instruction. Set the "changed" bit for the successor instruction.
 - If the successor instruction has been seen before, merge the operand stack and local variable values calculated in steps 2 and 3 into the values already there. Set the "changed" bit if there is any modification to the values.

5. Continue at step 1.

To merge two operand stacks, the number of values on each stack must be identical. The types of values on the stacks must also be identical, except that differently typed **reference** values may appear at corresponding places on the two stacks. In this case, the merged operand stack contains a **reference** to an instance of the first common superclass of the two types. Such a reference type always exists because the type `Object` is a superclass of all class and interface types. If the operand stacks cannot be merged, verification of the method fails.

To merge two local variable array states, corresponding pairs of local variables are compared. If the two types are not identical, then unless both contain **reference** values, the verifier records that the local variable contains an unusable value. If both of the pair of local variables contain **reference** values, the merged state contains a **reference** to an instance of the

first common superclass of the two types.

If the data-flow analyzer runs on a method without reporting a verification failure, then the method has been successfully verified by Pass 3 of the `class` file verifier.

Certain instructions and data types complicate the data-flow analyzer. We now examine each of these in more detail.

4.9.3 Values of Types `long` and `double`

Values of the `long` and `double` types are treated specially by the verification process.

Whenever a value of type `long` or `double` is moved into a local variable at index `n`, index `n + 1` is specially marked to indicate that it has been reserved by the value at index `n` and may not be used as a local variable index. Any value previously at index `n + 1` becomes unusable.

Whenever a value is moved to a local variable at index `n`, the index `n - 1` is examined to see if it is the index of a value of type `long` or `double`. If so, the local variable at index `n - 1` is changed to indicate that it now contains an unusable value. Since the local variable at index `n` has been overwritten, the local variable at index `n - 1` cannot represent a value of type `long` or `double`.

Dealing with values of types `long` or `double` on the operand stack is simpler; the verifier treats them as single values on the stack. For example, the verification code for the *dadd* opcode (add two `double` values) checks that the top two items on the stack are both of type `double`. When calculating operand stack length, values of type `long` and `double` have length two.

Untyped instructions that manipulate the operand stack must treat values of type `double` and `long` as atomic (indivisible). For example, the verifier reports a failure if the top value on the stack is a `double` and it encounters an instruction such as *pop* or *dup*. The instructions *pop2* or *dup2* must be used instead.

4.9.4 Instance Initialization Methods and Newly Created Objects

Creating a new class instance is a multistep process. The statement

```
...
new MyClass(i, j, k);
...
```

can be implemented by the following:

```
...
```

```

        new #1                // Allocate uninitialized
space for myClass
        dup                  // Duplicate object on the
operand stack
        iload_1              // Push i
        iload_2              // Push j
        iload_3              // Push k
        invokespecial #5     // Invoke myClass.<init>
        ...

```

This instruction sequence leaves the newly created and initialized object on top of the operand stack. (Additional examples of compilation to the instruction set of the Java virtual machine are given in [Chapter 7, "Compiling for the Java Virtual Machine."](#))

The instance initialization method ([§3.9](#)) for class `myClass` sees the new uninitialized object as its `this` argument in local variable `0`. Before that method invokes another instance initialization method of `myClass` or its direct superclass on `this`, the only operation the method can perform on `this` is assigning fields declared within `myClass`.

When doing dataflow analysis on instance methods, the verifier initializes local variable `0` to contain an object of the current class, or, for instance initialization methods, local variable `0` contains a special type indicating an uninitialized object. After an appropriate instance initialization method is invoked (from the current class or the current superclass) on this object, all occurrences of this special type on the verifier's model of the operand stack and in the local variable array are replaced by the current class type. The verifier rejects code that uses the new object before it has been initialized or that initializes the object more than once. In addition, it ensures that every normal return of the method has invoked an instance initialization method either in the class of this method or in the direct superclass.

Similarly, a special type is created and pushed on the verifier's model of the operand stack as the result of the Java virtual machine instruction `new`. The special type indicates the instruction by which the class instance was created and the type of the uninitialized class instance created. When an instance initialization method is invoked on that class instance, all occurrences of the special type are replaced by the intended type of the class instance. This change in type may propagate to subsequent instructions as the dataflow analysis proceeds.

The instruction number needs to be stored as part of the special type, as there may be multiple not-yet-initialized instances of a class in existence on the operand stack at one time. For example, the Java virtual machine instruction sequence that implements

```
new InputStream(new Foo(), new InputStream("foo"))
```

may have two uninitialized instances of `InputStream` on the operand stack at once. When an instance initialization method is invoked on a class instance, only those occurrences of the special type on the operand stack or in the local variable array that are the *same object* as the

class instance are replaced.

A valid instruction sequence must not have an uninitialized object on the operand stack or in a local variable during a backwards branch, or in a local variable in code protected by an exception handler or a `finally` clause. Otherwise, a devious piece of code might fool the verifier into thinking it had initialized a class instance when it had, in fact, initialized a class instance created in a previous pass through a loop.

4.9.5 Exception Handlers

Java virtual machine code produced by Sun's compiler for the Java programming language always generates exception handlers such that:

- Either the ranges of instructions protected by two different exception handlers always are completely disjoint, or else one is a subrange of the other. There is never a partial overlap of ranges.
- The handler for an exception will never be inside the code that is being protected.
- The only entry to an exception handler is through an exception. It is impossible to fall through or "goto" the exception handler.

These restrictions are not enforced by the `class` file verifier since they do not pose a threat to the integrity of the Java virtual machine. As long as every nonexceptional path to the exception handler causes there to be a single object on the operand stack, and as long as all other criteria of the verifier are met, the verifier will pass the code.

4.9.6 Exceptions and `finally`

Given the code fragment

```
...
try {
    startFaucet();
    waterLawn();
} finally {
    stopFaucet();
}
...
```

the Java programming language guarantees that `stopFaucet` is invoked (the faucet is turned off) whether we finish watering the lawn or whether an exception occurs while starting the faucet or watering the lawn. That is, the `finally` clause is guaranteed to be executed whether its `try` clause completes normally or completes abruptly by throwing an exception.

To implement the `try-finally` construct, Sun's compiler for the Java programming language uses the exception-handling facilities together with two special instructions: *jsr* ("jump to subroutine") and *ret* ("return from subroutine"). The `finally` clause is compiled as a subroutine within the Java virtual machine code for its method, much like the code for an exception handler. When a *jsr* instruction that invokes the subroutine is executed, it pushes its return address, the address of the instruction after the *jsr* that is being executed, onto the operand stack as a value of type `returnAddress`. The code for the subroutine stores the return address in a local variable. At the end of the subroutine, a *ret* instruction fetches the return address from the local variable and transfers control to the instruction at the return address.

Control can be transferred to the `finally` clause (the `finally` subroutine can be invoked) in several different ways. If the `try` clause completes normally, the `finally` subroutine is invoked via a *jsr* instruction before evaluating the next expression. A `break` or `continue` inside the `try` clause that transfers control outside the `try` clause executes a *jsr* to the code for the `finally` clause first. If the `try` clause executes a `return`, the compiled code does the following:

1. Saves the return value (if any) in a local variable.
2. Executes a *jsr* to the code for the `finally` clause.
3. Upon return from the `finally` clause, returns the value saved in the local variable.

The compiler sets up a special exception handler, which catches any exception thrown by the `try` clause. If an exception is thrown in the `try` clause, this exception handler does the following:

1. Saves the exception in a local variable.
2. Executes a *jsr* to the `finally` clause.
3. Upon return from the `finally` clause, rethrows the exception.

For more information about the implementation of the `try-finally` construct, see [Section 7.13, "Compiling finally."](#)

The code for the `finally` clause presents a special problem to the verifier. Usually, if a particular instruction can be reached via multiple paths and a particular local variable contains incompatible values through those multiple paths, then the local variable becomes unusable. However, a `finally` clause might be called from several different places, yielding several different circumstances:

- The invocation from the exception handler may have a certain local variable that contains an exception.
- The invocation to implement `return` may have some local variable that contains the return value.
- The invocation from the bottom of the `try` clause may have an indeterminate value in that same local variable.

The code for the `finally` clause itself might pass verification, but after completing the updating all the successors of the `ret` instruction, the verifier would note that the local variable that the exception handler expects to hold an exception, or that the return code expects to hold a return value, now contains an indeterminate value.

Verifying code that contains a `finally` clause is complicated. The basic idea is the following:

- Each instruction keeps track of the list of `jsr` targets needed to reach that instruction. For most code, this list is empty. For instructions inside code for the `finally` clause, it is of length one. For multiply nested `finally` code (extremely rare!), it may be longer than one.
- For each instruction and each `jsr` needed to reach that instruction, a bit vector is maintained of all local variables accessed or modified since the execution of the `jsr` instruction.
- When executing the `ret` instruction, which implements a return from a subroutine, there must be only one possible subroutine from which the instruction can be returning. Two different subroutines cannot "merge" their execution to a single `ret` instruction.
- To perform the data-flow analysis on a `ret` instruction, a special procedure is used. Since the verifier knows the subroutine from which the instruction must be returning, it can find all the `jsr` instructions that call the subroutine and merge the state of the operand stack and local variable array at the time of the `ret` instruction into the operand stack and local variable array of the instructions following the `jsr`. Merging uses a special set of values for local variables:
 - For any local variable that the bit vector (constructed above) indicates has been accessed or modified by the subroutine, use the type of the local variable at the time of the `ret`.
 - For other local variables, use the type of the local variable before the `jsr` instruction.

4.10 Limitations of the Java Virtual Machine

The following limitations of the Java virtual machine are implicit in the `class` file format:

- The per-class or per-interface constant pool is limited to 65535 entries by the 16-bit `constant_pool_count` field of the `ClassFile` structure ([§4.1](#)). This acts as an internal limit on the total complexity of a single class or interface.

- The amount of code per non-native, non-abstract method is limited to 65536 bytes by the sizes of the indices in the `exception_table` of the `Code` attribute (§4.7.3), in the `LineNumberTable` attribute (§4.7.8), and in the `LocalVariableTable` attribute (§4.7.9).
- The greatest number of local variables in the local variables array of a frame created upon invocation of a method is limited to 65535 by the size of the `max_locals` item of the `Code` attribute (§4.7.3) giving the code of the method. Note that values of type `long` and `double` are each considered to reserve two local variables and contribute two units toward the `max_locals` value, so use of local variables of those types further reduces this limit.
- The number of fields that may be declared by a class or interface is limited to 65535 by the size of the `fields_count` item of the `ClassFile` structure (§4.1). Note that the value of the `fields_count` item of the `ClassFile` structure does not include fields that are inherited from superclasses or superinterfaces.
- The number of methods that may be declared by a class or interface is limited to 65535 by the size of the `methods_count` item of the `ClassFile` structure (§4.1). Note that the value of the `methods_count` item of the `ClassFile` structure does not include methods that are inherited from superclasses or superinterfaces.
- The number of direct superinterfaces of a class or interface is limited to 65535 by the size of the `interfaces_count` item of the `ClassFile` structure (§4.1).
- The size of an operand stack in a frame (§3.6) is limited to 65535 values by the `max_stack` field of the `Code_attribute` structure (§4.7.3). Note that values of type `long` and `double` are each considered to contribute two units toward the `max_stack` value, so use of values of these types on the operand stack further reduces this limit.
- The number of local variables in a frame (§3.6) is limited to 65535 by the `max_locals` field of the `Code_attribute` structure (§4.7.3) and the 16-bit local variable indexing of the Java virtual machine instruction set.
- The number of dimensions in an array is limited to 255 by the size of the *dimensions* opcode of the *multianewarray* instruction and by the constraints imposed on the *multianewarray*, *anewarray*, and *newarray* instructions by §4.8.2.
- The number of method parameters is limited to 255 by the definition of a method descriptor (§4.3.3), where the limit includes one unit for `this` in the case of instance or interface method invocations. Note that a method descriptor is defined in terms of a notion of method parameter length in which a parameter of type `long` or `double` contributes two units to the length, so parameters of these types further reduce the limit.

- The length of field and method names, field and method descriptors, and other constant string values is limited to 65535 characters by the 16-bit unsigned `length` item of the `CONSTANT_Utf8_info` structure (§4.4.7). Note that the limit is on the number of bytes in the encoding and not on the number of encoded characters. UTF-8 encodes some characters using two or three bytes. Thus, strings incorporating multibyte characters are further constrained.

¹ The Java virtual machine implementation of Sun's JDK release 1.0.2 supports `class` file format versions 45.0 through 45.3 inclusive. Sun's JDK releases 1.1.X can support `class` file formats of versions in the range 45.0 through 45.65535 inclusive. Implementations of version 1.2 of the Java 2 platform can support `class` file formats of versions in the range 45.0 through 46.0 inclusive.

² In retrospect, making 8-byte constants take two constant pool entries was a poor choice.

³ The first edition of *The Java Language Specification* required that "`com`" be in uppercase in this example. The second edition will reverse that convention and use lowercase.

⁴ The fact that `end_pc` is exclusive is a historical mistake in the design of the Java virtual machine: if the Java virtual machine code for a method is exactly 65535 bytes long and ends with an instruction that is 1 byte long, then that instruction cannot be protected by an exception handler. A compiler writer can work around this bug by limiting the maximum size of the generated Java virtual machine code for any method, instance initialization method, or static initializer (the size of any `code` array) to 65534 bytes.

⁵ The `InnerClasses` attribute was introduced in JDK release 1.1 to support nested classes and interfaces.

⁶ The `Synthetic` attribute was introduced in JDK release 1.1 to support nested classes and interfaces.

⁷ The `Deprecated` attribute was introduced in JDK release 1.1 to support the `@deprecated` tag in documentation comments.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The Java Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections through our [feedback form](#)

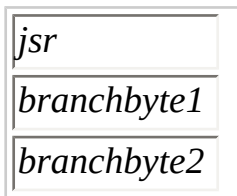
[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

jsr

Operation

Jump subroutine

Format



Forms

jsr = 168 (0xa8)

Operand Stack

... $\Rightarrow$..., *address*

Description

The *address* of the opcode of the instruction immediately following this *jsr* instruction is pushed onto the operand stack as a value of type `returnAddress`. The unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is $(branchbyte1 \ll 8) \mid branchbyte2$. Execution proceeds at that offset from the address of this *jsr* instruction. The target address must be that of an opcode of an instruction within the method that contains this *jsr* instruction.

Notes

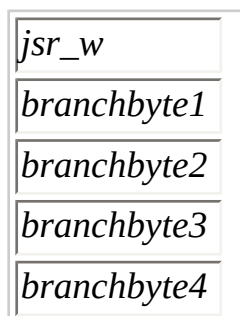
The *jsr* instruction is used with the *ret* instruction in the implementation of the `finally` clauses of the Java programming language (see [Section 7.13, "Compiling finally"](#)). Note that *jsr* pushes the address onto the operand stack and *ret* gets it out of a local variable. This asymmetry is intentional.

jsr_w

Operation

Jump subroutine (wide index)

Format



Forms

jsr_w = 201 (0xc9)

Operand Stack

... $\Rightarrow$..., *address*

Description

The *address* of the opcode of the instruction immediately following this *jsr_w* instruction is pushed onto the operand stack as a value of type `returnAddress`. The unsigned *branchbyte1*, *branchbyte2*, *branchbyte3*, and *branchbyte4* are used to construct a signed 32-bit offset, where the offset is $(branchbyte1 \ll 24) \mid (branchbyte2 \ll 16) \mid (branchbyte3 \ll 8) \mid branchbyte4$. Execution proceeds at that offset from the address of this *jsr_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *jsr_w* instruction.

Notes

The *jsr_w* instruction is used with the *ret* instruction in the implementation of the `finally` clauses of the Java programming language (see [Section 7.13, "Compiling finally"](#)). Note that *jsr_w* pushes the address onto the operand stack and *ret* gets it out of a local variable. This asymmetry is intentional.

Although the *jsr_w* instruction takes a 4-byte branch offset, other factors limit the size of a method to 65535 bytes ([§4.10](#)). This limit may be raised in a future release of the Java virtual machine.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The JavaTM Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections to jvm@java.sun.com

tableswitch

Operation

Access jump table by index and jump

Format

<i>tableswitch</i>
<0-3 byte pad \>
<i>defaultbyte1</i>
<i>defaultbyte2</i>
<i>defaultbyte3</i>
<i>defaultbyte4</i>
<i>lowbyte1</i>
<i>lowbyte2</i>
<i>lowbyte3</i>
<i>lowbyte4</i>
<i>highbyte1</i>
<i>highbyte2</i>
<i>highbyte3</i>
<i>highbyte4</i>
<i>jump offsets...</i>

Forms

tableswitch = 170 (0xaa)

Operand Stack

..., *index* $\Rightarrow$...

Description

A *tableswitch* is a variable-length instruction. Immediately after the *tableswitch* opcode, between 0 and 3 null bytes (zeroed bytes, not the null object) are inserted as padding. The number of null bytes is chosen so that the following byte begins at an address that is a multiple of 4 bytes from the start of the current method (the opcode of its first instruction). Immediately after the padding follow bytes constituting three signed 32-bit values: *default*, *low*, and *high*. Immediately following those bytes are bytes constituting a series of *high* - *low* + 1 signed 32-bit offsets. The value *low* must be less than or equal to *high*. The *high* - *low* + 1 signed 32-bit offsets are treated as a 0-based jump table. Each of these signed 32-bit values is constructed as $(\text{byte1} \ll 24) \mid (\text{byte2} \ll 16) \mid (\text{byte3} \ll 8) \mid \text{byte4}$.

The *index* must be of type `int` and is popped from the operand stack. If *index* is less than *low* or *index* is greater than *high*, then a target address is calculated by adding *default* to the address of the opcode of this *tableswitch* instruction. Otherwise, the offset at position *index* - *low* of the jump table is extracted. The target address is calculated by adding that offset to the address of the opcode of this *tableswitch* instruction. Execution then continues at the target address.

The target address that can be calculated from each jump table offset, as well as the ones that can be calculated from *default*, must be the address of an opcode of an instruction within the method that contains this *tableswitch* instruction.

Notes

The alignment required of the 4-byte operands of the *tableswitch* instruction guarantees 4-byte alignment of those operands if and only if the method that contains the *tableswitch* starts on a 4-byte boundary.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The JavaTM Virtual Machine Specification

[Copyright](#) © 1999 Sun Microsystems, Inc. All rights reserved

Please send any comments or corrections to jvm@java.sun.com

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

caload

Operation

Load char from array

Format

<i>caload</i>

Forms

caload = 52 (0x34)

Operand Stack

..., *arrayref*, *index* $\Rightarrow$..., *value*

Description

The *arrayref* must be of type **reference** and must refer to an array whose components are of type **char**. The *index* must be of type **int**. Both *arrayref* and *index* are popped from the operand stack. The component of the array at *index* is retrieved and zero-extended to an **int** *value*. That *value* is pushed onto the operand stack.

Runtime Exceptions

If *arrayref* is **null**, *caload* throws a **NullPointerException**.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *caload* instruction throws an **ArrayIndexOutOfBoundsException**.

castore

Operation

Store into `char` array

Format

`castore`

Forms

`castore = 85 (0x55)`

Operand Stack

`..., arrayref, index, value` $\Rightarrow$...

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `char`. The *index* and the *value* must both be of type `int`. The *arrayref*, *index*, and *value* are popped from the operand stack. The `int` *value* is truncated to a `char` and stored as the component of the array indexed by *index*.

Runtime Exceptions

If *arrayref* is `null`, *castore* throws a `NullPointerException`.

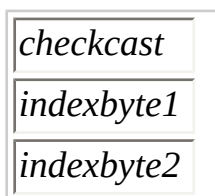
Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *castore* instruction throws an `ArrayIndexOutOfBoundsException`.

checkcast

Operation

Check whether object is of given type

Format



Forms

checkcast = 192 (0xc0)

Operand Stack

..., *objectref* $\Rightarrow$..., *objectref*

Description

The *objectref* must be of type **reference**. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class ([§3.6](#)), where the value of the index is $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$. The runtime constant pool item at the index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved ([§5.4.3.1](#)).

If *objectref* is `null` or can be cast to the resolved class, array, or interface type, the operand stack is unchanged; otherwise, the *checkcast* instruction throws a `ClassCastException`.

The following rules are used to determine whether an *objectref* that is not `null` can be cast to the resolved type: if S is the class of the object referred to by *objectref* and T is the resolved class, array, or interface type, *checkcast* determines whether *objectref* can be cast to type T as follows:

- If S is an ordinary (nonarray) class, then:
 - If T is a class type, then S must be the same class ([§2.8.1](#)) as T, or a subclass of T.
 - If T is an interface type, then S must implement ([§2.13](#)) interface T.
- If S is an interface type, then:
 - If T is a class type, then T must be `Object` ([§2.4.7](#)).
 - If T is an interface type, then T must be the same interface as S or a superinterface of S ([§2.13.2](#)).

- If *S* is a class representing the array type *SC*[], that is, an array of components of type *SC*, then:
 - If *T* is a class type, then *T* must be `Object` (§2.4.7).
 - If *T* is an array type *TC*[], that is, an array of components of type *TC*, then one of the following must be true:
 - *TC* and *SC* are the same primitive type (§2.4.1).
 - *TC* and *SC* are reference types (§2.4.6), and type *SC* can be cast to *TC* by recursive application of these rules.
 - If *T* is an interface type, *T* must be one of the interfaces implemented by arrays (§2.15).

Linking Exceptions

During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in [Section 5.4.3.1](#) can be thrown.

Runtime Exception

Otherwise, if *objectref* cannot be cast to the resolved class, array, or interface type, the *checkcast* instruction throws a `ClassCastException`.

Notes

The *checkcast* instruction is very similar to the *instanceof* instruction. It differs in its treatment of `null`, its behavior when its test fails (*checkcast* throws an exception, *instanceof* pushes a result code), and its effect on the operand stack.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The Java™ Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections to jvm@java.sun.com

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

new

Operation

Create new object

Format

<i>new</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

Forms

new = 187 (0xbb)

Operand Stack

... $\Rightarrow$..., *objectref*

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is $(indexbyte1 \ll 8) \mid indexbyte2$. The runtime constant pool item at the index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved (§5.4.3.1) and should result in a class type (it should not result in an array or interface type). Memory for a new instance of that class is allocated from the garbage-collected heap, and the instance variables of the new object are initialized to their default initial values (§2.5.1). The *objectref*, a reference to the instance, is pushed onto the operand stack.

On successful resolution of the class, it is initialized (§5.5) if it has not already been initialized.

Linking Exceptions

During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in [Section 5.4.3.1](#) can be thrown.

Otherwise, if the symbolic reference to the class, array, or interface type resolves to an interface or is an **abstract** class, *new* throws an **InstantiationError**.

Runtime Exception

Otherwise, if execution of this *new* instruction causes initialization of the referenced class, *new* may throw an **Error** as detailed in [Section 2.17.5](#).

Note

The *new* instruction does not completely create a new instance; instance creation is not completed until an instance initialization method has been invoked on the uninitialized instance.

newarray

Operation

Create new array

Format

<i>newarray</i>
<i>atype</i>

Forms

newarray = 188 (0xbc)

Operand Stack

..., *count* $\Rightarrow$..., *arrayref*

Description

The *count* must be of type `int`. It is popped off the operand stack. The *count* represents the number of elements in the array to be created.

The *atype* is a code that indicates the type of array to create. It must take one of the following values:

Array Type	<i>atype</i>
T_BOOLEAN	4
T_CHAR	5
T_FLOAT	6
T_DOUBLE	7
T_BYTE	8
T_SHORT	9
T_INT	10
T_LONG	11

A new array whose components are of type *atype* and of length *count* is allocated from the garbage-collected heap. A reference *arrayref* to this new array object is pushed into the operand stack. Each of the elements of the new array is initialized to the default initial value for the type of the array ([§2.5.1](#)).

Runtime Exception

If *count* is less than zero, *newarray* throws a `NegativeArraySizeException`.

Notes

In Sun's implementation of the Java virtual machine, arrays of type `boolean` (*atype* is `T_BOOLEAN`) are stored as arrays of 8-bit values and are manipulated using the *baload* and *bastore* instructions, instructions that also access arrays of type `byte`. Other implementations may implement packed `boolean` arrays; the *baload* and *bastore* instructions must still be used to access those arrays.

nop

Operation

Do nothing

Format



Forms

nop = 0 (0x0)

Operand Stack

No change

Description

Do nothing.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The JavaTM Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections to jvm@java.sun.com

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [L](#) [M](#) [N](#) [P](#) [R](#) [S](#) [T](#) [W](#)

monitorenter

Operation

Enter monitor for object

Format

monitorenter

Forms

monitorenter = 194 (0xc2)

Operand Stack

..., *objectref* $\Rightarrow$...

Description

The *objectref* must be of type `reference`.

Each object has a monitor associated with it. The thread that executes *monitorenter* gains ownership of the monitor associated with *objectref*. If another thread already owns the monitor associated with *objectref*, the current thread waits until the object is unlocked, then tries again to gain ownership. If the current thread already owns the monitor associated with *objectref*, it increments a counter in the monitor indicating the number of times this thread has entered the monitor. If the monitor associated with *objectref* is not owned by any thread, the current thread becomes the owner of the monitor, setting the entry count of this monitor to 1.

Runtime Exception

If *objectref* is `null`, *monitorenter* throws a `NullPointerException`.

Notes

For detailed information about threads and monitors in the Java virtual machine, see [Chapter 8, "Threads and Locks."](#)

A *monitorenter* instruction may be used with one or more *monitorexit* instructions to implement a **synchronized** statement in the Java programming language. The *monitorenter* and *monitorexit* instructions are not used in the implementation of **synchronized** methods, although they can be used to provide equivalent locking semantics; however, monitor entry on invocation of a **synchronized** method is handled implicitly by the Java virtual machine's method invocation instructions. See [Section 7.14](#) for more information on the use of the *monitorenter* and *monitorexit* instructions.

The association of a monitor with an object may be managed in various ways that are beyond the scope of this specification. For instance, the monitor may be allocated and deallocated at the same time as the object. Alternatively, it may be dynamically allocated at the time when a thread attempts to gain exclusive access to the object and freed at some later time when no thread remains in the monitor for the object.

The synchronization constructs of the Java programming language require support for operations on monitors besides entry and exit. These include waiting on a monitor (`Object.wait`) and notifying other threads waiting on a monitor (`Object.notifyAll` and `Object.notify`). These operations are supported in the standard package `java.lang` supplied with the Java virtual machine. No explicit support for these operations appears in the instruction set of the Java virtual machine.

monitorexit

Operation

Exit monitor for object

Format

<i>monitorexit</i>

Forms

monitorexit = 195 (0xc3)

Operand Stack

..., *objectref* $\Rightarrow$...

Description

The *objectref* must be of type `reference`.

The current thread should be the owner of the monitor associated with the instance referenced by *objectref*. The thread decrements the counter indicating the number of times it has entered this monitor. If as a result the value of the counter becomes zero, the current thread releases the monitor. If the monitor associated with *objectref* becomes free, other threads that are waiting to acquire that monitor are allowed to attempt to do so.

Runtime Exceptions

If *objectref* is `null`, *monitorexit* throws a `NullPointerException`.

Otherwise, if the current thread is not the owner of the monitor, *monitorexit* throws an `IllegalMonitorStateException`.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in [Section 8.13](#) and if the second of those rules is violated by the execution of this *monitorexit* instruction, then *monitorexit* throws an `IllegalMonitorStateException`.

Notes

For detailed information about threads and monitors in the Java virtual machine, see [Chapter 8, "Threads and Locks."](#)

One or more *monitorexit* instructions may be used with a *monitorenter* instruction to implement a `synchronized` statement in the Java programming language. The *monitorenter* and *monitorexit* instructions are not used in the implementation of `synchronized` methods, although they can be used to provide equivalent locking semantics.

The Java virtual machine supports exceptions thrown within `synchronized` methods and `synchronized` statements differently. Monitor exit on normal `synchronized` method completion is handled by the Java virtual machine's return instructions. Monitor exit on abrupt `synchronized` method completion is handled implicitly by the Java virtual machine's *athrow* instruction. When an exception is thrown from within a `synchronized` statement, exit from the monitor entered prior

to the execution of the `synchronized` statement is achieved using the Java virtual machine's exception handling mechanism. See [Section 7.14](#) for more information on the use of the *monitorenter* and *monitorexit* instructions.

multianewarray

Operation

Create new multidimensional array

Format

<i>multianewarray</i>
<i>indexbyte1</i>
<i>indexbyte2</i>
<i>dimensions</i>

Forms

multianewarray = 197 (0xc5)

Operand Stack

..., *count1*, [*count2*, ...] $\Rightarrow$ *...*, *arrayref*

Description

The *dimensions* operand is an unsigned byte that must be greater than or equal to 1. It represents the number of dimensions of the array to be created. The operand stack must contain *dimensions* values. Each such value represents the number of components in a dimension of the array to be created, must be of type `int`, and must be nonnegative. The *count1* is the desired length in the first dimension, *count2* in the second, etc.

All of the *count* values are popped off the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class ([§3.6](#)), where the value of the index is $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$. The runtime constant pool item at the index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved ([§5.4.3.1](#)). The resulting entry must be an array class type of dimensionality greater than or equal to

dimensions.

A new multidimensional array of the array type is allocated from the garbage-collected heap. If any *count* value is zero, no subsequent dimensions are allocated. The components of the array in the first dimension are initialized to subarrays of the type of the second dimension, and so on. The components of the last allocated dimension of the array are initialized to the default initial value for the type of the components (§2.5.1). A reference *arrayref* to the new array is pushed onto the operand stack.

Linking Exceptions

During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in [Section 5.4.3.1](#) can be thrown.

Otherwise, if the current class does not have permission to access the element type of the resolved array class, *multianewarray* throws an `IllegalAccessException`.

Runtime Exception

Otherwise, if any of the *dimensions* values on the operand stack are less than zero, the *multianewarray* instruction throws a `NegativeArraySizeException`.

Notes

It may be more efficient to use *newarray* or *anewarray* when creating an array of a single dimension.

The array class referenced via the runtime constant pool may have more dimensions than the *dimensions* operand of the *multianewarray* instruction. In that case, only the first *dimensions* of the dimensions of the array are created.

[Contents](#) | [Prev](#) | [Next](#) | [Index](#)

The JavaTM Virtual Machine Specification

[Copyright © 1999 Sun Microsystems, Inc.](#) All rights reserved

Please send any comments or corrections to jvm@java.sun.com

[Overview](#) [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

Java™ 2 Platform
Std. Ed. v1.4.2

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | FIELD | [CONSTR](#) | [METHOD](#)

DETAIL: FIELD | [CONSTR](#) | METHOD

java.lang

Class RuntimeException

[java.lang.Object](#)

extended by [java.lang.Throwable](#)

extended by [java.lang.Exception](#)

extended by [java.lang.RuntimeException](#)

All Implemented Interfaces:

[Serializable](#)

Direct Known Subclasses:

[ArithmeticException](#), [ArrayStoreException](#), [BufferOverflowException](#),
[BufferUnderflowException](#), [CannotRedoException](#), [CannotUndoException](#),
[ClassCastException](#), [CMMException](#), [ConcurrentModificationException](#), [DOMException](#),
[EmptyStackException](#), [IllegalArgumentException](#), [IllegalMonitorStateException](#),
[IllegalPathStateException](#), [IllegalStateException](#), [ImagingOpException](#),
[IndexOutOfBoundsException](#), [MissingResourceException](#), [NegativeArraySizeException](#),
[NoSuchElementException](#), [NullPointerException](#), [ProfileDataException](#), [ProviderException](#),
[RasterFormatException](#), [SecurityException](#), [SystemException](#),
[UndeclaredThrowableException](#), [UnmodifiableSetException](#), [UnsupportedOperationException](#)

public class **RuntimeException**

extends [Exception](#)

RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.

A method is not required to declare in its **throws** clause any subclasses of **RuntimeException** that might be thrown during the execution of the method but not caught.

Since:

JDK1.0

See Also:

[Serialized Form](#)

Constructor Summary

[RuntimeException](#)()

Constructs a new runtime exception with `null` as its detail message.

[RuntimeException](#)([String](#) message)

Constructs a new runtime exception with the specified detail message.

[RuntimeException](#)([String](#) message, [Throwable](#) cause)

Constructs a new runtime exception with the specified detail message and cause.

[RuntimeException](#)([Throwable](#) cause)

Constructs a new runtime exception with the specified cause and a detail message of `(cause==null ? null : cause.toString())` (which typically contains the class and detail message of cause).

Methods inherited from class `java.lang.Throwable`

[fillInStackTrace](#), [getCause](#), [getLocalizedMessage](#), [getMessage](#), [getStackTrace](#), [initCause](#), [printStackTrace](#), [printStackTrace](#), [printStackTrace](#), [setStackTrace](#), [toString](#)

Methods inherited from class `java.lang.Object`

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [wait](#), [wait](#), [wait](#)

Constructor Detail

RuntimeException

```
public RuntimeException()
```

Constructs a new runtime exception with `null` as its detail message. The cause is not initialized, and may subsequently be initialized by a call to [Throwable.initCause](#) (`java.lang.Throwable`).

RuntimeException

```
public RuntimeException(String message)
```

Constructs a new runtime exception with the specified detail message. The cause is not initialized, and may subsequently be initialized by a call to [Throwable.initCause\(\[java.lang.Throwable\]\(#\)\)](#).

Parameters:

`message` - the detail message. The detail message is saved for later retrieval by the [Throwable.getMessage\(\)](#) method.

RuntimeException

```
public RuntimeException(String message,  
                      Throwable cause)
```

Constructs a new runtime exception with the specified detail message and cause.

Note that the detail message associated with `cause` is *not* automatically incorporated in this runtime exception's detail message.

Parameters:

`message` - the detail message (which is saved for later retrieval by the [Throwable.getMessage\(\)](#) method).

`cause` - the cause (which is saved for later retrieval by the [Throwable.getCause\(\)](#) method). (A `null` value is permitted, and indicates that the cause is nonexistent or unknown.)

Since:

1.4

RuntimeException

```
public RuntimeException(Throwable cause)
```

Constructs a new runtime exception with the specified cause and a detail message of `(cause==null ? null : cause.toString())` (which typically contains the class and detail message of `cause`). This constructor is useful for runtime exceptions that are little more than wrappers for other throwables.

Parameters:

cause - the cause (which is saved for later retrieval by the [Throwable.getCause\(\)](#) method). (A `null` value is permitted, and indicates that the cause is nonexistent or unknown.)

Since:

1.4

[Overview](#) [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.4.2*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | FIELD | [CONSTR](#) | [METHOD](#) DETAIL: FIELD | [CONSTR](#) | METHOD

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)*Java™ 2 Platform
Std. Ed. v1.4.2*[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#) [All Classes](#)SUMMARY: NESTED | FIELD | [CONSTR](#) | [METHOD](#) DETAIL: FIELD | [CONSTR](#) | METHOD

java.lang

Class ClassCastException

[java.lang.Object](#)└ [java.lang.Throwable](#)└ [java.lang.Exception](#)└ [java.lang.RuntimeException](#)└ **java.lang.ClassCastException**

All Implemented Interfaces:

[Serializable](#)

public class **ClassCastException**extends [RuntimeException](#)

Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance. For example, the following code generates a `ClassCastException`:

```
Object x = new Integer(0);
System.out.println((String)x);
```

Since:

JDK1.0

See Also:

[Serialized Form](#)

Constructor Summary

[ClassCastException](#)()Constructs a `ClassCastException` with no detail message.

[ClassCastException](#)([String](#) s)Constructs a `ClassCastException` with the specified detail message.**Methods inherited from class java.lang.[Throwable](#)**[fillInStackTrace](#), [getCause](#), [getLocalizedMessage](#), [getMessage](#), [getStackTrace](#), [initCause](#), [printStackTrace](#), [printStackTrace](#), [printStackTrace](#), [setStackTrace](#), [toString](#)**Methods inherited from class java.lang.[Object](#)**[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [wait](#), [wait](#), [wait](#)

Constructor Detail

ClassCastException

public **ClassCastException**()Constructs a `ClassCastException` with no detail message.

ClassCastException

public **ClassCastException**([String](#) s)Constructs a `ClassCastException` with the specified detail message.**Parameters:**

s - the detail message.

[Overview](#) **[Package](#)** **[Class](#)** **[Use Tree](#)** **[Deprecated](#)** **[Index](#)** **[Help](#)****[PREV CLASS](#)** **[NEXT CLASS](#)****[FRAMES](#)** **[NO FRAMES](#)** **[All Classes](#)**SUMMARY: NESTED | FIELD | [CONSTR](#) | [METHOD](#) DETAIL: FIELD | [CONSTR](#) | METHOD*Java™ 2 Platform
Std. Ed. v1.4.2*[Submit a bug or feature](#)For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That

documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).



Jasmin Home Page

Jon Meyer

March 1997

Jasmin is a Java Assembler Interface. It takes ASCII descriptions for Java classes, written in a simple assembler-like syntax and using the Java Virtual Machine instruction set. It converts them into binary Java class files suitable for loading into a JVM implementation.

The *Java Virtual Machine* is a platform-neutral runtime engine used to execute Java programs. The JVM is generating a lot of interest these days because of its secure, efficient multi-threaded architecture. People are now developing compilers for other programming languages that generate code which can be run on the Java Virtual Machine (see my Resource Guide below).



[Java Virtual Machine](#)

Jasmin was written as the companion to the book "Java Virtual Machine", published by O'Reilly, written by Troy Downing and myself. (available now!) (see [cover](#)).

Documentation

[About Jasmin.](#)

Describes what Jasmin is, why I wrote Jasmin, and the kinds of people who might be interested in Jasmin.

[Jasmin User Guide](#)

A basic Jasmin user guide. It describes the syntax of the assembler, and how to use it.

[Jasmin README](#)

The Jasmin README file that comes with the Jasmin distribution.

[Jasmin FAQ](#)

Gives email feedback on Jasmin, and Q+A on the JVM in general.

[Java Virtual Machine Instruction Reference](#)

A quick reference for all of the JVM instructions. Includes Jasmin syntax.

Related Links

See my [Resource Guide](#) for a more comprehensive list of online things to do with the Java Virtual Machine.

Jasmin uses [Java Cup](#), and [JAS](#) internally, so you should check out these pages too.

Downloading Jasmin

You can obtain Jasmin 1.0 from:

- [jasmin-1.06.tar.gz](#) (approx 410K)
- [jasmin.zip](#) (approx 600K)

This version of Jasmin is written in Java, and needs Sun's JDK 1.0 to run or equivalent. I have tested it under Solaris, SGI and Windows 95 (using Symantec Cafe and Visual J++).

For people who use systems with filename restrictions, a copy of the ZIP file containing the classes used by Jasmin is available [here](#).

Questions, Problems

If you have any questions or problems, you can email me at: meyer_jon@yahoo.com.

Disassemblers

The following JVM disassemblers support Jasmin format:

D-Java

- Shawn Silverman has created a disassembler that can output files in the Jasmin syntax. This lets you take a Java class file, disassemble it, modify the VM code, and reassemble it. Versions for Windows and Solaris are available. See [D-Java](#).

DJava (Java version)

- **NEW!** Shawn Silverman has also created a new pure Java implementation of D-Java, which includes a "builder" that lets you assemble, generate, and disassemble .class files and provides a Jasmin output format. You can download this from [Shawn Silverman's FTP site](#), or see: [Shawn Silverman's pages](#).

JavaClass API

- **NEW!** Markus Dahm has created the [JavaClass API](#), which also provides a Java API to .class files, and includes the ability to assemble, generate, and disassemble .class files and provides a Jasmin

output format. There is also a simple Mini language which uses the JavaClass API.

Kimera Disassembler

- The folks at [Kimera](#) have developed an online disassembler which spits out Jasmin.

Alternatives

Jason Hunt (djh4@cs.wustl.edu) has written a Java assembler, which you may want to check out. See <http://siesta.cs.wustl.edu/~djh4/>).

[Java Grinder](#) is another Java assembler/disassembler with a text based syntax.

Copyright (1998) Jonathan Meyer, under the terms of the GNU General Public License.

Work on Jasmin is supported in part by the [NYU Media Research Laboratory](#).

[Jasmin Home](#) | [Jon Meyer's Home](#)

java.lang

Class Class<T>

[java.lang.Object](#)└─ [java.lang.Class<T>](#)

All Implemented Interfaces:

[Serializable](#), [AnnotatedElement](#), [GenericDeclaration](#), [Type](#)

```
public final class Class<T>
```

extends [Object](#)implements [Serializable](#), [GenericDeclaration](#), [Type](#), [AnnotatedElement](#)

Instances of the class `Class` represent classes and interfaces in a running Java application. An enum is a kind of class and an annotation is a kind of interface. Every array also belongs to a class that is reflected as a `Class` object that is shared by all arrays with the same element type and number of dimensions. The primitive Java types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`), and the keyword `void` are also represented as `Class` objects.

`Class` has no public constructor. Instead `Class` objects are constructed automatically by the Java Virtual Machine as classes are loaded and by calls to the `defineClass` method in the class loader.

The following example uses a `Class` object to print the class name of an object:

```
void printClassName(Object obj) {
    System.out.println("The class of " + obj +
        " is " + obj.getClass().
getName());
}
```

It is also possible to get the `Class` object for a named type (or for `void`) using a class literal (JLS Section [15.8.2](#)). For example:

```
System.out.println("The name of class Foo is: "+Foo.
```

```
class.getName());
```

Since:

JDK1.0

See Also:
[ClassLoader.defineClass\(byte\[\], int, int\)](#), [Serialized Form](#)

Method Summary

<U> Class <? extends U>	asSubclass (Class <U> clazz) Casts this <code>Class</code> object to represent a subclass of the class represented by the specified class object.
T	cast (Object obj) Casts an object to the class or interface represented by this <code>Class</code> object.
boolean	desiredAssertionStatus () Returns the assertion status that would be assigned to this class if it were to be initialized at the time this method is invoked.
static Class <?>	forName (String className) Returns the <code>Class</code> object associated with the class or interface with the given string name.
static Class <?>	forName (String name, boolean initialize, ClassLoader loader) Returns the <code>Class</code> object associated with the class or interface with the given string name, using the given class loader.
<A extends Annotation > A	getAnnotation (Class <A> annotationClass) Returns the annotation of the specified type that is present on this element.
Annotation []	getAnnotations () Returns all annotations present on this element.
Class []	getClasses () Returns an array containing <code>Class</code> objects representing all the public classes and interfaces that are members of the class represented by this <code>Class</code> object.
ClassLoader	getClassLoader () Returns the class loader for the class.

Class <?>	getComponentType () Returns the Class representing the component type of an array.
Constructor <T>	getConstructor (Class ... parameterTypes) Returns a Constructor object that reflects the specified public constructor of the class represented by this Class object.
Constructor []	getConstructors () Returns an array containing Constructor objects reflecting all the public constructors of the class represented by this Class object.
Annotation []	getDeclaredAnnotations () Returns all annotations that are directly present on this element.
Class []	getDeclaredClasses () Returns an array of Class objects reflecting all the classes and interfaces declared as members of the class represented by this Class object.
Constructor <T>	getDeclaredConstructor (Class ... parameterTypes) Returns a Constructor object that reflects the specified constructor of the class or interface represented by this Class object.
Constructor []	getDeclaredConstructors () Returns an array of Constructor objects reflecting all the constructors declared by the class represented by this Class object.
Field	getDeclaredField (String name) Returns a Field object that reflects the specified declared field of the class or interface represented by this Class object.
Field []	getDeclaredFields () Returns an array of Field objects reflecting all the fields declared by the class or interface represented by this Class object.
Method	getDeclaredMethod (String name, Class ... parameterTypes) Returns a Method object that reflects the specified declared method of the class or interface represented by this Class object.

Method []	getDeclaredMethods () Returns an array of Method objects reflecting all the methods declared by the class or interface represented by this Class object.
Class <?>	getDeclaringClass () If the class or interface represented by this Class object is a member of another class, returns the Class object representing the class in which it was declared.
Constructor <?>	getEnclosingConstructor () If this Class object represents a local or anonymous class within a constructor, returns a Constructor object representing the immediately enclosing constructor of the underlying class.
Method	getEnclosingMethod () If this Class object represents a local or anonymous class within a method, returns a Method object representing the immediately enclosing method of the underlying class.
Enum []	getEnumConstants () Returns the elements of this enum class or null if this Class object does not represent an enum type.
Field	getField (String name) Returns a Field object that reflects the specified public member field of the class or interface represented by this Class object.
Field []	getFields () Returns an array containing Field objects reflecting all the accessible public fields of the class or interface represented by this Class object.
Type []	getGenericInterfaces () Returns the Types representing the interfaces directly implemented by the class or interface represented by this object.
Type	getGenericSuperclass () Returns the Type representing the direct superclass of the entity (class, interface, primitive type or void) represented by this Class .
Class []	getInterfaces () Determines the interfaces implemented by the class or interface represented by this object.

Method	getMethod (String name, Class ... parameterTypes) Returns a Method object that reflects the specified public member method of the class or interface represented by this Class object.
Method []	getMethods () Returns an array containing Method objects reflecting all the public <i>member</i> methods of the class or interface represented by this Class object, including those declared by the class or interface and those inherited from superclasses and superinterfaces.
int	getModifiers () Returns the Java language modifiers for this class or interface, encoded in an integer.
String	getName () Returns the name of the entity (class, interface, array class, primitive type, or void) represented by this Class object, as a String .
Package	getPackage () Gets the package for this class.
ProtectionDomain	getProtectionDomain () Returns the ProtectionDomain of this class.
URL	getResource (String name) Finds a resource with a given name.
InputStream	getResourceAsStream (String name) Finds a resource with a given name.
Object []	getSigners () Gets the signers of this class.
Class <? super T >	getSuperclass () Returns the Class representing the superclass of the entity (class, interface, primitive type or void) represented by this Class .
TypeVariable < Class < T >>[]	getTypeParameters () Returns an array of TypeVariable objects that represent the type variables declared by the generic declaration represented by this GenericDeclaration object, in declaration order.
boolean	isAnnotation () Returns true if this Class object represents an annotation type.

boolean	isAnnotationPresent (Class <? extends Annotation > annotationClass) Returns true if an annotation for the specified type is present on this element, else false.
boolean	isArray () Determines if this <code>Class</code> object represents an array class.
boolean	isAssignableFrom (Class <?> cls) Determines if the class or interface represented by this <code>Class</code> object is either the same as, or is a superclass or superinterface of, the class or interface represented by the specified <code>Class</code> parameter.
boolean	isEnum () Returns true if and only if this class was declared as an enum in the source code.
boolean	isInstance (Object obj) Determines if the specified <code>Object</code> is assignment-compatible with the object represented by this <code>Class</code> .
boolean	isInterface () Determines if the specified <code>Class</code> object represents an interface type.
boolean	isPrimitive () Determines if the specified <code>Class</code> object represents a primitive type.
boolean	isSynthetic () Returns true if this class is a synthetic class; returns false otherwise.
T	newInstance () Creates a new instance of the class represented by this <code>Class</code> object.
String	toString () Converts the object to a string.

Methods inherited from class [java.lang.Object](#)

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [wait](#), [wait](#), [wait](#)

Methods inherited from interface [java.lang.reflect.AnnotatedElement](#)

[getAnnotation](#)

Method Detail

toString

```
public String toString()
```

Converts the object to a string. The string representation is the string "class" or "interface", followed by a space, and then by the fully qualified name of the class in the format returned by `getName`. If this `Class` object represents a primitive type, this method returns the name of the primitive type. If this `Class` object represents void this method returns "void".

Overrides:

[toString](#) in class [Object](#)

Returns:

a string representation of this class object.

forName

```
public static Class<?> forName(String className)  
                                throws ClassNotFoundException
```

Returns the `Class` object associated with the class or interface with the given string name. Invoking this method is equivalent to:

```
Class.forName(className, true, currentLoader)
```

where `currentLoader` denotes the defining class loader of the current class.

For example, the following code fragment returns the runtime `Class` descriptor for the class named `java.lang.Thread`:

```
Class t = Class.forName("java.lang.Thread")
```

A call to `forName("X")` causes the class named X to be initialized.

Parameters:

`className` - the fully qualified name of the desired class.

Returns:

the `Class` object for the class with the specified name.

Throws:

[LinkageError](#) - if the linkage fails

[ExceptionInInitializerError](#) - if the initialization provoked by this method fails

[ClassNotFoundException](#) - if the class cannot be located

forName

```
public static Class<?> forName(String name,
                                boolean initialize,
                                ClassLoader loader)
    throws ClassNotFoundException
```

Returns the `Class` object associated with the class or interface with the given string name, using the given class loader. Given the fully qualified name for a class or interface (in the same format returned by `getName`) this method attempts to locate, load, and link the class or interface. The specified class loader is used to load the class or interface. If the parameter `loader` is null, the class is loaded through the bootstrap class loader. The class is initialized only if the `initialize` parameter is `true` and if it has not been initialized earlier.

If `name` denotes a primitive type or void, an attempt will be made to locate a user-defined class in the unnamed package whose name is `name`. Therefore, this method cannot be used to obtain any of the `Class` objects representing primitive types or void.

If `name` denotes an array class, the component type of the array class is loaded but not initialized.

For example, in an instance method the expression:

```
Class.forName("Foo")
```

is equivalent to:

```
Class.forName("Foo", true, this.getClass().
getClassLoader())
```

Note that this method throws errors related to loading, linking or initializing as specified in Sections 12.2, 12.3 and 12.4 of *The Java Language Specification*. Note that this method does not check whether the requested class is accessible to its caller.

If the `loader` is `null`, and a security manager is present, and the caller's class loader is not `null`, then this method calls the security manager's `checkPermission` method with a `RuntimePermission("getClassLoader")` permission to ensure it's ok to access the bootstrap class loader.

Parameters:

`name` - fully qualified name of the desired class
`initialize` - whether the class must be initialized
`loader` - class loader from which the class must be loaded

Returns:

class object representing the desired class

Throws:

[LinkageError](#) - if the linkage fails
[ExceptionInInitializerError](#) - if the initialization provoked by this method fails
[ClassNotFoundException](#) - if the class cannot be located by the specified class loader

Since:

1.2

See Also:

[forName\(String\)](#), [ClassLoader](#)

newInstance

```
public T newInstance()
    throws InstantiationException,
           IllegalAccessException
```

Creates a new instance of the class represented by this `Class` object. The class is instantiated as if by a `new` expression with an empty argument list. The class is initialized if it has not already been initialized.

Returns:

a newly allocated instance of the class represented by this object.

Throws:

[IllegalAccessException](#) - if the class or its nullary constructor is not accessible.
[InstantiationException](#) - if this `Class` represents an abstract class, an

interface, an array class, a primitive type, or void; or if the class has no nullary constructor; or if the instantiation fails for some other reason.

[ExceptionInInitializerError](#) - if the initialization provoked by this method fails.

[SecurityException](#) - If a security manager, *s*, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.PUBLIC\)](#) denies creation of new instances of this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies access to the package of this class

isInstance

```
public boolean isInstance(Object obj)
```

Determines if the specified `Object` is assignment-compatible with the object represented by this `Class`. This method is the dynamic equivalent of the Java language `instanceof` operator. The method returns `true` if the specified `Object` argument is non-null and can be cast to the reference type represented by this `Class` object without raising a `ClassCastException`. It returns `false` otherwise.

Specifically, if this `Class` object represents a declared class, this method returns `true` if the specified `Object` argument is an instance of the represented class (or of any of its subclasses); it returns `false` otherwise. If this `Class` object represents an array class, this method returns `true` if the specified `Object` argument can be converted to an object of the array class by an identity conversion or by a widening reference conversion; it returns `false` otherwise. If this `Class` object represents an interface, this method returns `true` if the class or any superclass of the specified `Object` argument implements this interface; it returns `false` otherwise. If this `Class` object represents a primitive type, this method returns `false`.

Parameters:

`obj` - the object to check

Returns:

`true` if `obj` is an instance of this class

Since:

JDK1.1

isAssignableFrom

```
public boolean isAssignableFrom(Class<?> cls)
```

Determines if the class or interface represented by this `Class` object is either the same as, or is a superclass or superinterface of, the class or interface represented by the specified `Class` parameter. It returns `true` if so; otherwise it returns `false`. If this `Class` object represents a primitive type, this method returns `true` if the specified `Class` parameter is exactly this `Class` object; otherwise it returns `false`.

Specifically, this method tests whether the type represented by the specified `Class` parameter can be converted to the type represented by this `Class` object via an identity conversion or via a widening reference conversion. See *The Java Language Specification*, sections 5.1.1 and 5.1.4, for details.

Parameters:

`cls` - the `Class` object to be checked

Returns:

the `boolean` value indicating whether objects of the type `cls` can be assigned to objects of this class

Throws:

[NullPointerException](#) - if the specified `Class` parameter is null.

Since:

JDK1.1

isInterface

```
public boolean isInterface()
```

Determines if the specified `Class` object represents an interface type.

Returns:

`true` if this object represents an interface; `false` otherwise.

isArray

```
public boolean isArray()
```

Determines if this `Class` object represents an array class.

Returns:

`true` if this object represents an array class; `false` otherwise.

Since:

JDK1.1

isPrimitive

```
public boolean isPrimitive()
```

Determines if the specified `Class` object represents a primitive type.

There are nine predefined `Class` objects to represent the eight primitive types and void. These are created by the Java Virtual Machine, and have the same names as the primitive types that they represent, namely `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`.

These objects may only be accessed via the following public static final variables, and are the only `Class` objects for which this method returns `true`.

Returns:

`true` if and only if this class represents a primitive type

Since:

JDK1.1

See Also:

[`Boolean.TYPE`](#), [`Character.TYPE`](#), [`Byte.TYPE`](#), [`Short.TYPE`](#), [`Integer.TYPE`](#), [`Long.TYPE`](#), [`Float.TYPE`](#), [`Double.TYPE`](#), [`Void.TYPE`](#)

isAnnotation

```
public boolean isAnnotation()
```

Returns `true` if this `Class` object represents an annotation type. Note that if this method returns `true`, [`isInterface\(\)`](#) would also return `true`, as all annotation types are also interfaces.

Returns:

`true` if this class object represents an annotation type; `false` otherwise

Since:

1.5

isSynthetic

```
public boolean isSynthetic()
```

Returns `true` if this class is a synthetic class; returns `false` otherwise.

Returns:

`true` if and only if this class is a synthetic class as defined by the Java Language Specification.

Since:

1.5

getName

```
public String getName()
```

Returns the name of the entity (class, interface, array class, primitive type, or void) represented by this `Class` object, as a `String`.

If this class object represents a reference type that is not an array type then the binary name of the class is returned, as specified by the Java Language Specification, Second Edition.

If this class object represents a primitive type or void, then the name returned is a `String` equal to the Java language keyword corresponding to the primitive type or void.

If this class object represents a class of arrays, then the internal form of the name consists of the name of the element type preceded by one or more '[' characters representing the depth of the array nesting. The encoding of element type names is as follows:

Element Type	Encoding
boolean	Z
byte	B
char	C
class or interface	<i>Lclassname;</i>
double	D
float	F
int	I
long	J

short

S

The class or interface name *classname* is the binary name of the class specified above.

Examples:

```
String.class.getName()
    returns "java.lang.String"
byte.class.getName()
    returns "byte"
(new Object[3]).getClass().getName()
    returns "[Ljava.lang.Object;"
(new int[3][4][5][6][7][8][9]).getClass().getName()
    returns "[[[[[[[I"
```

Returns:

the name of the class or interface represented by this object.

getClassLoader

public [ClassLoader](#) **getClassLoader()**

Returns the class loader for the class. Some implementations may use null to represent the bootstrap class loader. This method will return null in such implementations if this class was loaded by the bootstrap class loader.

If a security manager is present, and the caller's class loader is not null and the caller's class loader is not the same as or an ancestor of the class loader for the class whose class loader is requested, then this method calls the security manager's `checkPermission` method with a `RuntimePermission("getClassLoader")` permission to ensure it's ok to access the class loader for the class.

If this object represents a primitive type or void, null is returned.

Returns:

the class loader that loaded the class or interface represented by this object.

Throws:

[SecurityException](#) - if a security manager exists and its `checkPermission` method denies access to the class loader for the class.

See Also:

[ClassLoader](#), [SecurityManager.checkPermission\(java.security.](#)

[Permission](#)), [RuntimePermission](#)

getTypeParameters

```
public TypeVariable<Class<T>>[] getTypeParameters()
```

Returns an array of `TypeVariable` objects that represent the type variables declared by the generic declaration represented by this `GenericDeclaration` object, in declaration order. Returns an array of length 0 if the underlying generic declaration declares no type variables.

Specified by:

[getTypeParameters](#) in interface [GenericDeclaration](#)

Returns:

an array of `TypeVariable` objects that represent the type variables declared by this generic declaration

Throws:

`GenericSignatureFormatError` - if the generic signature of this generic declaration does not conform to the format specified in the Java Virtual Machine Specification, 3rd edition

Since:

1.5

getSuperclass

```
public Class<? super T> getSuperclass()
```

Returns the `Class` representing the superclass of the entity (class, interface, primitive type or void) represented by this `Class`. If this `Class` represents either the `Object` class, an interface, a primitive type, or void, then null is returned. If this object represents an array class then the `Class` object representing the `Object` class is returned.

Returns:

the superclass of the class represented by this object.

getGenericSuperclass

```
public Type getGenericSuperclass()
```

Returns the **Type** representing the direct superclass of the entity (class, interface, primitive type or void) represented by this **Class**.

If the superclass is a parameterized type, the **Type** object returned must accurately reflect the actual type parameters used in the source code. The parameterized type representing the superclass is created if it had not been created before. See the declaration of [ParameterizedType](#) for the semantics of the creation process for parameterized types. If this **Class** represents either the **Object** class, an interface, a primitive type, or void, then null is returned. If this object represents an array class then the **Class** object representing the **Object** class is returned.

Returns:

the superclass of the class represented by this object

Throws:

GenericSignatureFormatError - if the generic class signature does not conform to the format specified in the Java Virtual Machine Specification, 3rd edition

[TypeNotPresentException](#) - if the generic superclass refers to a non-existent type declaration

MalformedParameterizedTypeException - if the generic superclass refers to a parameterized type that cannot be instantiated for any reason

Since:

1.5

getPackage

```
public Package getPackage()
```

Gets the package for this class. The class loader of this class is used to find the package. If the class was loaded by the bootstrap class loader the set of packages loaded from CLASSPATH is searched to find the package of the class. Null is returned if no package object was created by the class loader of this class.

Packages have attributes for versions and specifications only if the information was defined in the manifests that accompany the classes, and if the class loader created the package instance with the attributes from the manifest.

Returns:

the package of the class, or null if no package information is available from the archive or codebase.

getInterfaces

```
public Class[] getInterfaces()
```

Determines the interfaces implemented by the class or interface represented by this object.

If this object represents a class, the return value is an array containing objects representing all interfaces implemented by the class. The order of the interface objects in the array corresponds to the order of the interface names in the `implements` clause of the declaration of the class represented by this object. For example, given the declaration:

```
class Shimmer implements FloorWax, DessertTopping
{ ... }
```

suppose the value of `s` is an instance of `Shimmer`; the value of the expression:

```
s.getClass().getInterfaces()[0]
```

is the `Class` object that represents interface `FloorWax`; and the value of:

```
s.getClass().getInterfaces()[1]
```

is the `Class` object that represents interface `DessertTopping`.

If this object represents an interface, the array contains objects representing all interfaces extended by the interface. The order of the interface objects in the array corresponds to the order of the interface names in the `extends` clause of the declaration of the interface represented by this object.

If this object represents a class or interface that implements no interfaces, the method returns an array of length 0.

If this object represents a primitive type or void, the method returns an array of length 0.

Returns:

an array of interfaces implemented by this class.

getGenericInterfaces

```
public Type[] getGenericInterfaces()
```

Returns the **Types** representing the interfaces directly implemented by the class or interface represented by this object.

If a superinterface is a parameterized type, the **Type** object returned for it must accurately reflect the actual type parameters used in the source code. The parameterized type representing each superinterface is created if it had not been created before. See the declaration of [ParameterizedType](#) for the semantics of the creation process for parameterized types.

If this object represents a class, the return value is an array containing objects representing all interfaces implemented by the class. The order of the interface objects in the array corresponds to the order of the interface names in the **implements** clause of the declaration of the class represented by this object. In the case of an array class, the interfaces **Cloneable** and **Serializable** are returned in that order.

If this object represents an interface, the array contains objects representing all interfaces directly extended by the interface. The order of the interface objects in the array corresponds to the order of the interface names in the **extends** clause of the declaration of the interface represented by this object.

If this object represents a class or interface that implements no interfaces, the method returns an array of length 0.

If this object represents a primitive type or void, the method returns an array of length 0.

Returns:

an array of interfaces implemented by this class

Throws:

GenericSignatureFormatError - if the generic class signature does not conform to the format specified in the Java Virtual Machine Specification, 3rd edition
[TypeNotPresentException](#) - if any of the generic superinterfaces refers to a non-existent type declaration
MalformedParameterizedTypeException - if any of the generic superinterfaces refer to a parameterized type that cannot be instantiated for any reason

Since:

1.5

getComponentType

```
public Class<?> getComponentType()
```

Returns the `Class` representing the component type of an array. If this class does not represent an array class this method returns null.

Returns:

the `Class` representing the component type of this class if this class is an array

Since:

JDK1.1

See Also:

[Array](#)

getModifiers

```
public int getModifiers()
```

Returns the Java language modifiers for this class or interface, encoded in an integer. The modifiers consist of the Java Virtual Machine's constants for `public`, `protected`, `private`, `final`, `static`, `abstract` and `interface`; they should be decoded using the methods of class `Modifier`.

If the underlying class is an array class, then its `public`, `private` and `protected` modifiers are the same as those of its component type. If this `Class` represents a primitive type or void, its `public` modifier is always `true`, and its `protected` and `private` modifiers are always `false`. If this object represents an array class, a primitive type or void, then its `final` modifier is always `true` and its interface modifier is always `false`. The values of its other modifiers are not determined by this specification.

The modifier encodings are defined in *The Java Virtual Machine Specification*, table 4.1.

Returns:

the `int` representing the modifiers for this class

Since:

JDK1.1

See Also:

[Modifier](#)

getSigners

```
public Object[] getSigners()
```

Gets the signers of this class.

Returns:

the signers of this class, or null if there are no signers. In particular, this method returns null if this object represents a primitive type or void.

Since:

JDK1.1

getEnclosingMethod

```
public Method getEnclosingMethod()
```

If this `Class` object represents a local or anonymous class within a method, returns a [Method](#) object representing the immediately enclosing method of the underlying class. Returns `null` otherwise. In particular, this method returns `null` if the underlying class is a local or anonymous class immediately enclosed by a type declaration, instance initializer or static initializer.

Returns:

the immediately enclosing method of the underlying class, if that class is a local or anonymous class; otherwise `null`.

Since:

1.5

getEnclosingConstructor

```
public Constructor<?> getEnclosingConstructor()
```

If this `Class` object represents a local or anonymous class within a constructor, returns a [Constructor](#) object representing the immediately enclosing constructor of the underlying class. Returns `null` otherwise. In particular, this method returns `null` if the underlying class is a local or anonymous class immediately enclosed by a type declaration, instance initializer or static initializer.

Returns:

the immediately enclosing constructor of the underlying class, if that class is a local or anonymous class; otherwise `null`.

Since:

1.5

getDeclaringClass

```
public Class<?> getDeclaringClass()
```

If the class or interface represented by this `Class` object is a member of another class, returns the `Class` object representing the class in which it was declared. This method returns null if this class or interface is not a member of any other class. If this `Class` object represents an array class, a primitive type, or void, then this method returns null.

Returns:

the declaring class for this class

Since:

JDK1.1

getClasses

```
public Class[] getClasses()
```

Returns an array containing `Class` objects representing all the public classes and interfaces that are members of the class represented by this `Class` object. This includes public class and interface members inherited from superclasses and public class and interface members declared by the class. This method returns an array of length 0 if this `Class` object has no public member classes or interfaces. This method also returns an array of length 0 if this `Class` object represents a primitive type, an array class, or void.

Returns:

the array of `Class` objects representing the public members of this class

Throws:

[SecurityException](#) - If a security manager, `s`, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.PUBLIC\)](#) method denies access to the classes within this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies access to the package of this class

Since:

JDK1.1

getFields


```
public Field[] getFields()
           throws SecurityException
```

Returns an array containing `Field` objects reflecting all the accessible public fields of the class or interface represented by this `Class` object. The elements in the array returned are not sorted and are not in any particular order. This method returns an array of length 0 if the class or interface has no accessible public fields, or if it represents an array class, a primitive type, or void.

Specifically, if this `Class` object represents a class, this method returns the public fields of this class and of all its superclasses. If this `Class` object represents an interface, this method returns the fields of this interface and of all its superinterfaces.

The implicit length field for array class is not reflected by this method. User code should use the methods of class `Array` to manipulate arrays.

See *The Java Language Specification*, sections 8.2 and 8.3.

Returns:

the array of `Field` objects representing the public fields

Throws:

[SecurityException](#) - If a security manager, `s`, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.PUBLIC\)](#) denies access to the fields within this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies access to the package of this class

Since:

JDK1.1

getMethods

```
public Method[] getMethods()
           throws SecurityException
```

Returns an array containing `Method` objects reflecting all the public *member* methods of the class or interface represented by this `Class` object, including those declared by the class or interface and those inherited from superclasses and superinterfaces. Array classes return all the member methods inherited from the `Object` class. The elements in the array returned are not sorted and are not in any particular order. This method returns an array of length 0 if this

Class object represents a class or interface that has no public member methods, or if this **Class** object represents a primitive type or void.

The class initialization method `<clinit>` is not included in the returned array. If the class declares multiple public member methods with the same parameter types, they are all included in the returned array.

See *The Java Language Specification*, sections 8.2 and 8.4.

Returns:

the array of **Method** objects representing the public methods of this class

Throws:

[SecurityException](#) - If a security manager, *s*, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.PUBLIC\)](#) denies access to the methods within this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies access to the package of this class

Since:

JDK1.1

getConstructors

```
public Constructor[] getConstructors()
                        throws SecurityException
```

Returns an array containing **Constructor** objects reflecting all the public constructors of the class represented by this **Class** object. An array of length 0 is returned if the class has no public constructors, or if the class is an array class, or if the class reflects a primitive type or void.

Returns:

the array containing **Method** objects for all the declared public constructors of this class matches the specified **parameterTypes**

Throws:

[SecurityException](#) - If a security manager, *s*, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.PUBLIC\)](#) denies access to the constructors within this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies

access to the package of this class

Since:

JDK1.1

getField

```
public Field getField(String name)
    throws NoSuchFieldException,
           SecurityException
```

Returns a `Field` object that reflects the specified public member field of the class or interface represented by this `Class` object. The `name` parameter is a `String` specifying the simple name of the desired field.

The field to be reflected is determined by the algorithm that follows. Let `C` be the class represented by this object:

1. If `C` declares a public field with the name specified, that is the field to be reflected.
2. If no field was found in step 1 above, this algorithm is applied recursively to each direct superinterface of `C`. The direct superinterfaces are searched in the order they were declared.
3. If no field was found in steps 1 and 2 above, and `C` has a superclass `S`, then this algorithm is invoked recursively upon `S`. If `C` has no superclass, then a `NoSuchFieldException` is thrown.

See *The Java Language Specification*, sections 8.2 and 8.3.

Parameters:

`name` - the field name

Returns:

the `Field` object of this class specified by `name`

Throws:

[NoSuchFieldException](#) - if a field with the specified name is not found.

[NullPointerException](#) - if `name` is `null`

[SecurityException](#) - If a security manager, `s`, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.PUBLIC\)](#) denies access to the field
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies access to the package of this class

Since:

getMethod

```
public Method getMethod(String name,  
                        Class... parameterTypes)  
    throws NoSuchMethodException,  
           SecurityException
```

Returns a `Method` object that reflects the specified public member method of the class or interface represented by this `Class` object. The `name` parameter is a `String` specifying the simple name the desired method. The `parameterTypes` parameter is an array of `Class` objects that identify the method's formal parameter types, in declared order. If `parameterTypes` is `null`, it is treated as if it were an empty array.

If the `name` is "<init>" or "<clinit>" a `NoSuchMethodException` is raised. Otherwise, the method to be reflected is determined by the algorithm that follows. Let `C` be the class represented by this object:

1. `C` is searched for any *matching methods*. If no matching method is found, the algorithm of step 1 is invoked recursively on the superclass of `C`.
2. If no method was found in step 1 above, the superinterfaces of `C` are searched for a matching method. If any such method is found, it is reflected.

To find a matching method in a class `C`: If `C` declares exactly one public method with the specified name and exactly the same formal parameter types, that is the method reflected. If more than one such method is found in `C`, and one of these methods has a return type that is more specific than any of the others, that method is reflected; otherwise one of the methods is chosen arbitrarily.

See *The Java Language Specification*, sections 8.2 and 8.4.

Parameters:

`name` - the name of the method

`parameterTypes` - the list of parameters

Returns:

the `Method` object that matches the specified name and `parameterTypes`

Throws:

[NoSuchMethodException](#) - if a matching method is not found or if the name is "<init>" or "<clinit>".

[NullPointerException](#) - if `name` is `null`

[SecurityException](#) - If a security manager, `s`, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.PUBLIC\)](#) denies access to the method
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies access to the package of this class

Since:

JDK1.1

getConstructor

```
public Constructor<T> getConstructor(Class... parameterTypes)
                                throws NoSuchMethodException,
                                       SecurityException
```

Returns a `Constructor` object that reflects the specified public constructor of the class represented by this `Class` object. The `parameterTypes` parameter is an array of `Class` objects that identify the constructor's formal parameter types, in declared order.

The constructor to reflect is the public constructor of the class represented by this `Class` object whose formal parameter types match those specified by `parameterTypes`.

Parameters:

`parameterTypes` - the parameter array

Returns:

the `Method` object of the public constructor that matches the specified `parameterTypes`

Throws:

[NoSuchMethodException](#) - if a matching method is not found.

[SecurityException](#) - If a security manager, `s`, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.PUBLIC\)](#) denies access to the constructor
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies access to the package of this class

Since:

JDK1.1

getDeclaredClasses

```
public Class[] getDeclaredClasses()
           throws SecurityException
```

Returns an array of `Class` objects reflecting all the classes and interfaces declared as members of the class represented by this `Class` object. This includes public, protected, default (package) access, and private classes and interfaces declared by the class, but excludes inherited classes and interfaces. This method returns an array of length 0 if the class declares no classes or interfaces as members, or if this `Class` object represents a primitive type, an array class, or void.

Returns:

the array of `Class` objects representing all the declared members of this class

Throws:

[SecurityException](#) - If a security manager, *s*, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.DECLARED\)](#) denies access to the declared classes within this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies access to the package of this class

Since:

JDK1.1

getDeclaredFields

```
public Field[] getDeclaredFields()
           throws SecurityException
```

Returns an array of `Field` objects reflecting all the fields declared by the class or interface represented by this `Class` object. This includes public, protected, default (package) access, and private fields, but excludes inherited fields. The elements in the array returned are not sorted and are not in any particular order. This method returns an array of length 0 if the class or interface declares no fields, or if this `Class` object represents a primitive type, an array class, or void.

See *The Java Language Specification*, sections 8.2 and 8.3.

Returns:

the array of `Field` objects representing all the declared fields of this class

Throws:

[SecurityException](#) - If a security manager, *s*, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.DECLARED\)](#) denies access to the declared fields within this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies access to the package of this class

Since:

JDK1.1

getDeclaredMethods

```
public Method[] getDeclaredMethods()
                               throws SecurityException
```

Returns an array of `Method` objects reflecting all the methods declared by the class or interface represented by this `Class` object. This includes public, protected, default (package) access, and private methods, but excludes inherited methods. The elements in the array returned are not sorted and are not in any particular order. This method returns an array of length 0 if the class or interface declares no methods, or if this `Class` object represents a primitive type, an array class, or void. The class initialization method `<clinit>` is not included in the returned array. If the class declares multiple public member methods with the same parameter types, they are all included in the returned array.

See *The Java Language Specification*, section 8.2.

Returns:

the array of `Method` objects representing all the declared methods of this class

Throws:

[SecurityException](#) - If a security manager, `s`, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.DECLARED\)](#) denies access to the declared methods within this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies access to the package of this class

Since:

JDK1.1

getDeclaredConstructors

```
public Constructor[] getDeclaredConstructors()
```

throws [SecurityException](#)

Returns an array of `Constructor` objects reflecting all the constructors declared by the class represented by this `Class` object. These are public, protected, default (package) access, and private constructors. The elements in the array returned are not sorted and are not in any particular order. If the class has a default constructor, it is included in the returned array. This method returns an array of length 0 if this `Class` object represents an interface, a primitive type, an array class, or void.

See *The Java Language Specification*, section 8.2.

Returns:

the array of `Method` objects representing all the declared constructors of this class

Throws:

[SecurityException](#) - If a security manager, `s`, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.DECLARED\)](#) denies access to the declared constructors within this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies access to the package of this class

Since:

JDK1.1

getDeclaredField

```
public Field getDeclaredField(String name)
                               throws NoSuchFieldException,
                                       SecurityException
```

Returns a `Field` object that reflects the specified declared field of the class or interface represented by this `Class` object. The `name` parameter is a `String` that specifies the simple name of the desired field. Note that this method will not reflect the `length` field of an array class.

Parameters:

`name` - the name of the field

Returns:

the `Field` object for the specified field in this class

Throws:

[NoSuchFieldException](#) - if a field with the specified name is not found.

[NullPointerException](#) - if `name` is `null`

[SecurityException](#) - If a security manager, *s*, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.DECLARED\)](#) denies access to the declared field
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies access to the package of this class

Since:

JDK1.1

getDeclaredMethod

```
public Method getDeclaredMethod(String name,
                                Class... parameterTypes)
    throws NoSuchMethodException,
           SecurityException
```

Returns a `Method` object that reflects the specified declared method of the class or interface represented by this `Class` object. The `name` parameter is a `String` that specifies the simple name of the desired method, and the `parameterTypes` parameter is an array of `Class` objects that identify the method's formal parameter types, in declared order. If more than one method with the same parameter types is declared in a class, and one of these methods has a return type that is more specific than any of the others, that method is returned; otherwise one of the methods is chosen arbitrarily. If the name is "<init>" or "<clinit>" a `NoSuchMethodException` is raised.

Parameters:

`name` - the name of the method

`parameterTypes` - the parameter array

Returns:

the `Method` object for the method of this class matching the specified name and parameters

Throws:

[NoSuchMethodException](#) - if a matching method is not found.

[NullPointerException](#) - if `name` is `null`

[SecurityException](#) - If a security manager, *s*, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.DECLARED\)](#) denies access to the declared method
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies access to the package of this class

Since:

JDK1.1

getDeclaredConstructor

```
public Constructor<T> getDeclaredConstructor  
(Class... parameterTypes)
```

throws [NoSuchMethodException](#),
[SecurityException](#)

Returns a `Constructor` object that reflects the specified constructor of the class or interface represented by this `Class` object. The `parameterTypes` parameter is an array of `Class` objects that identify the constructor's formal parameter types, in declared order.

Parameters:

`parameterTypes` - the parameter array

Returns:

The `Method` object for the constructor with the specified parameter list

Throws:

[NoSuchMethodException](#) - if a matching method is not found.

[SecurityException](#) - If a security manager, `s`, is present and any of the following conditions is met:

- invocation of [s.checkMemberAccess\(this, Member.DECLARED\)](#) denies access to the declared constructor
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of [s.checkPackageAccess\(\)](#) denies access to the package of this class

Since:

JDK1.1

getResourceAsStream

```
public InputStream getResourceAsStream(String name)
```

Finds a resource with a given name. The rules for searching resources associated with a given class are implemented by the defining [class loader](#) of the class. This method delegates to this object's class loader. If this object was loaded by the bootstrap class loader, the method delegates to [ClassLoader.getResourceAsStream\(java.lang.String\)](#).

Before delegation, an absolute resource name is constructed from the given resource name using this algorithm:

- If the name begins with a '/' ('\\u002f'), then the absolute name of the resource is the portion of the name following the '/'.
- Otherwise, the absolute name is of the following form:

modified_package_name/name

Where the `modified_package_name` is the package name of this object with '/' substituted for '.' ('\\u002e').

Parameters:

name - name of the desired resource

Returns:

A [InputStream](#) object or null if no resource with this name is found

Throws:

[NullPointerException](#) - If name is null

Since:

JDK1.1

getResource

```
public URL getResource(String name)
```

Finds a resource with a given name. The rules for searching resources associated with a given class are implemented by the defining [class loader](#) of the class. This method delegates to this object's class loader. If this object was loaded by the bootstrap class loader, the method delegates to [ClassLoader.getResource\(java.lang.String\)](#).

Before delegation, an absolute resource name is constructed from the given resource name using this algorithm:

- If the name begins with a '/' ('\\u002f'), then the absolute name of the resource is the portion of the name following the '/'.
- Otherwise, the absolute name is of the following form:

modified_package_name/name

Where the `modified_package_name` is the package name of this object with '/'

substituted for ' . ' (' \u002e ').

Parameters:

name - name of the desired resource

Returns:

A [URL](#) object or null if no resource with this name is found

Since:

JDK1.1

getProtectionDomain

```
public ProtectionDomain getProtectionDomain()
```

Returns the `ProtectionDomain` of this class. If there is a security manager installed, this method first calls the security manager's `checkPermission` method with a `RuntimePermission("getProtectionDomain")` permission to ensure it's ok to get the `ProtectionDomain`.

Returns:

the `ProtectionDomain` of this class

Throws:

[SecurityException](#) - if a security manager exists and its `checkPermission` method doesn't allow getting the `ProtectionDomain`.

Since:

1.2

See Also:

[ProtectionDomain](#), [SecurityManager.checkPermission\(java.security.Permission\)](#), [RuntimePermission](#)

desiredAssertionStatus

```
public boolean desiredAssertionStatus()
```

Returns the assertion status that would be assigned to this class if it were to be initialized at the time this method is invoked. If this class has had its assertion status set, the most recent setting will be returned; otherwise, if any package default assertion status pertains to this class, the most recent setting for the most specific pertinent package default assertion status is returned; otherwise, if this class is not a system class (i.e., it has a class loader) its class loader's default assertion status is returned; otherwise, the system class default assertion status is returned.

Few programmers will have any need for this method; it is provided for the benefit of the JRE itself. (It allows a class to determine at the time that it is initialized whether assertions should be enabled.) Note that this method is not guaranteed to return the actual assertion status that was (or will be) associated with the specified class when it was (or will be) initialized.

Returns:

the desired assertion status of the specified class.

Since:

1.4

See Also:

[ClassLoader.setClassAssertionStatus\(java.lang.String, boolean\)](#), [ClassLoader.setPackageAssertionStatus\(java.lang.String, boolean\)](#), [ClassLoader.setDefaultAssertionStatus\(boolean\)](#)

isEnum

```
public boolean isEnum()
```

Returns true if and only if this class was declared as an enum in the source code.

Returns:

true if and only if this class was declared as an enum in the source code

getEnumConstants

```
public T[] getEnumConstants()
```

Returns the elements of this enum class or null if this Class object does not represent an enum type.

Returns:

an array containing the values comprising the enum class represented by this Class object in the order they're declared, or null if this Class object does not represent an enum type

cast

```
public T cast(Object obj)
```

Casts an object to the class or interface represented by this `Class` object.

Parameters:

`obj` - the object to be cast

Returns:

the object after casting, or null if `obj` is null

Throws:

[ClassCastException](#) - if the object is not null and is not assignable to the type `T`.

Since:

1.5

asSubclass

```
public <U> Class<? extends U> asSubclass(Class<U> clazz)
```

Casts this `Class` object to represent a subclass of the class represented by the specified class object. Checks that that the cast is valid, and throws a `ClassCastException` if it is not. If this method succeeds, it always returns a reference to this class object.

This method is useful when a client needs to "narrow" the type of a `Class` object to pass it to an API that restricts the `Class` objects that it is willing to accept. A cast would generate a compile-time warning, as the correctness of the cast could not be checked at runtime (because generic types are implemented by erasure).

Throws:

[ClassCastException](#) - if this `Class` object does not represent a subclass of the specified class (here "subclass" includes the class itself).

getAnnotation

```
public <A extends Annotation> A getAnnotation  
(Class<A> annotationClass)
```

isAnnotationPresent

```
public boolean isAnnotationPresent(Class<? extends
Annotation> annotationClass)
```

Description copied from interface: [AnnotatedElement](#)

Returns true if an annotation for the specified type is present on this element, else false. This method is designed primarily for convenient access to marker annotations.

Specified by:

[isAnnotationPresent](#) in interface [AnnotatedElement](#)

Parameters:

annotationClass - the Class object corresponding to the annotation type

Returns:

true if an annotation for the specified annotation type is present on this element, else false

getAnnotations

```
public Annotation[] getAnnotations()
```

Description copied from interface: [AnnotatedElement](#)

Returns all annotations present on this element. (Returns an array of length zero if this element has no annotations.) The caller of this method is free to modify the returned array; it will have no effect on the arrays returned to other callers.

Specified by:

[getAnnotations](#) in interface [AnnotatedElement](#)

Returns:

all annotations present on this element

getDeclaredAnnotations

```
public Annotation[] getDeclaredAnnotations()
```

Description copied from interface: [AnnotatedElement](#)

Returns all annotations that are directly present on this element. Unlike the other methods in this interface, this method ignores inherited annotations. (Returns an array of length zero if no annotations are directly present on this element.) The caller of this method is free to modify the returned array; it will have no effect on the arrays returned to other callers.

Specified by:[getDeclaredAnnotations](#) in interface [AnnotatedElement](#)**Returns:**

All annotations directly present on this element

[Overview](#) **[Package](#)** **[Class](#)** **[Use Tree](#)** **[Deprecated](#)** **[Index](#)** **[Help](#)***JavaTM 2 Platform*[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#)[NO FRAMES](#)[All Classes](#)*Std. Ed. v1.5.0*SUMMARY: NESTED | FIELD | CONSTR | [METHOD](#) DETAIL: FIELD | CONSTR | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

java.lang

Class String

[java.lang.Object](#)└─ [java.lang.String](#)

All Implemented Interfaces:

[Serializable](#), [CharSequence](#), [Comparable](#)<[String](#)>

```
public final class String
```

extends [Object](#)implements [Serializable](#), [Comparable](#)<[String](#)>, [CharSequence](#)

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};  
String str = new String(data);
```

Here are some more examples of how strings can be used:

```
System.out.println("abc");  
String cde = "cde";  
System.out.println("abc" + cde);  
String c = "abc".substring(2,3);  
String d = cde.substring(1, 2);
```

The class `String` includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase. Case mapping is based on the Unicode Standard version specified by the [Character](#) class.

The Java language provides special support for the string concatenation operator (`+`), and for conversion of other objects to strings. String concatenation is implemented through the `StringBuffer` class and its `append` method. String conversions are implemented through the method `toString`, defined by `Object` and inherited by all classes in Java. For additional information on string concatenation and conversion, see Gosling, Joy, and Steele, *The Java Language Specification*.

Unless otherwise noted, passing a `null` argument to a constructor or method in this class will cause a [NullPointerException](#) to be thrown.

A `String` represents a string in the UTF-16 format in which *supplementary characters* are represented by *surrogate pairs* (see the section [Unicode Character Representations](#) in the `Character` class for more information). Index values refer to `char` code units, so a supplementary character uses two positions in a `String`.

The `String` class provides methods for dealing with Unicode code points (i.e., characters), in addition to those for dealing with Unicode code units (i.e., `char` values).

Since:

JDK1.0

See Also:

[Object.toString\(\)](#), [StringBuffer](#), [StringBuffer.append\(boolean\)](#), [StringBuffer.append\(char\)](#), [StringBuffer.append\(char\[\]\)](#), [StringBuffer.append\(char\[\], int, int\)](#), [StringBuffer.append\(double\)](#), [StringBuffer.append\(float\)](#), [StringBuffer.append\(int\)](#), [StringBuffer.append\(long\)](#), [StringBuffer.append\(java.lang.Object\)](#), [StringBuffer.append\(java.lang.String\)](#), [Charset](#), [Serialized Form](#)

Field Summary

static [Comparator](#)<[String](#)>

[CASE_INSENSITIVE_ORDER](#)

A `Comparator` that orders `String` objects as by `compareToIgnoreCase`.

Constructor Summary

[String](#)()

Initializes a newly created `String` object so that it represents an empty character sequence.

[String](#)(byte[] bytes)

Constructs a new `String` by decoding the specified array of bytes using the platform's default charset.

[String](#)(byte[] ascii, int hibyte)

Deprecated. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a charset name or that use the platform's default charset.*

[String](#)(byte[] bytes, int offset, int length)

Constructs a new `String` by decoding the specified subarray of bytes using the platform's default charset.

[String](#)(byte[] ascii, int hibyte, int offset, int count)

Deprecated. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a charset name or that use the platform's default charset.*

[String](#)(byte[] bytes, int offset, int length, [String](#) charsetName)

Constructs a new `String` by decoding the specified subarray of bytes using the specified charset.

[String](#)(byte[] bytes, [String](#) charsetName)

Constructs a new `String` by decoding the specified array of bytes using the specified charset.

[String](#)(char[] value)

Allocates a new `String` so that it represents the sequence of characters currently contained in the character array argument.

[String](#)(char[] value, int offset, int count)

Allocates a new `String` that contains characters from a subarray of the character array argument.

[String](#)(int[] codePoints, int offset, int count)

Allocates a new `String` that contains characters from a subarray of the Unicode code point array argument.

[String](#)([String](#) original)

Initializes a newly created `String` object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

[String](#)([StringBuffer](#) buffer)

Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

[String](#)([StringBuilder](#) builder)

Allocates a new string that contains the sequence of characters currently contained in the string builder argument.

Method Summary

char	charAt (int index) Returns the char value at the specified index.
int	codePointAt (int index) Returns the character (Unicode code point) at the specified index.
int	codePointBefore (int index) Returns the character (Unicode code point) before the specified index.
int	codePointCount (int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this String.
int	compareTo (String anotherString) Compares two strings lexicographically.
int	compareToIgnoreCase (String str) Compares two strings lexicographically, ignoring case differences.
String	concat (String str) Concatenates the specified string to the end of this string.
boolean	contains (CharSequence s) Returns true if and only if this string contains the specified sequence of char values.
boolean	contentEquals (CharSequence cs) Returns true if and only if this String represents the same sequence of char values as the specified sequence.
boolean	contentEquals (StringBuffer sb) Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
static String	copyValueOf (char[] data) Returns a String that represents the character sequence in the array specified.
static String	copyValueOf (char[] data, int offset, int count) Returns a String that represents the character sequence in the array specified.
boolean	endsWith (String suffix) Tests if this string ends with the specified suffix.

boolean	<code>equals</code> (<code>Object</code> anObject) Compares this string to the specified object.
boolean	<code>equalsIgnoreCase</code> (<code>String</code> anotherString) Compares this <code>String</code> to another <code>String</code> , ignoring case considerations.
static <code>String</code>	<code>format</code> (<code>Locale</code> l, <code>String</code> format, <code>Object</code> ... args) Returns a formatted string using the specified locale, format string, and arguments.
static <code>String</code>	<code>format</code> (<code>String</code> format, <code>Object</code> ... args) Returns a formatted string using the specified format string and arguments.
byte[]	<code>getBytes</code> () Encodes this <code>String</code> into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
void	<code>getBytes</code> (int srcBegin, int srcEnd, byte[] dst, int dstBegin) Deprecated. <i>This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the <code>getBytes()</code> method, which uses the platform's default charset.</i>
byte[]	<code>getBytes</code> (<code>String</code> charsetName) Encodes this <code>String</code> into a sequence of bytes using the named charset, storing the result into a new byte array.
void	<code>getChars</code> (int srcBegin, int srcEnd, char[] dst, int dstBegin) Copies characters from this string into the destination character array.
int	<code>hashCode</code> () Returns a hash code for this string.
int	<code>indexOf</code> (int ch) Returns the index within this string of the first occurrence of the specified character.
int	<code>indexOf</code> (int ch, int fromIndex) Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
int	<code>indexOf</code> (<code>String</code> str) Returns the index within this string of the first occurrence of the specified substring.
int	<code>indexOf</code> (<code>String</code> str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

String	intern () Returns a canonical representation for the string object.
int	lastIndexOf (int ch) Returns the index within this string of the last occurrence of the specified character.
int	lastIndexOf (int ch, int fromIndex) Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
int	lastIndexOf (String str) Returns the index within this string of the rightmost occurrence of the specified substring.
int	lastIndexOf (String str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
int	length () Returns the length of this string.
boolean	matches (String regex) Tells whether or not this string matches the given regular expression .
int	offsetByCodePoints (int index, int codePointOffset) Returns the index within this String that is offset from the given index by codePointOffset code points.
boolean	regionMatches (boolean ignoreCase, int toffset, String other, int ooffset, int len) Tests if two string regions are equal.
boolean	regionMatches (int toffset, String other, int ooffset, int len) Tests if two string regions are equal.
String	replace (char oldChar, char newChar) Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
String	replace (CharSequence target, CharSequence replacement) Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
String	replaceAll (String regex, String replacement) Replaces each substring of this string that matches the given regular expression with the given replacement.

String	replaceFirst (String regex, String replacement) Replaces the first substring of this string that matches the given regular expression with the given replacement.
String[]	split (String regex) Splits this string around matches of the given regular expression.
String[]	split (String regex, int limit) Splits this string around matches of the given regular expression.
boolean	startsWith (String prefix) Tests if this string starts with the specified prefix.
boolean	startsWith (String prefix, int toffset) Tests if this string starts with the specified prefix beginning a specified index.
CharSequence	subSequence (int beginIndex, int endIndex) Returns a new character sequence that is a subsequence of this sequence.
String	substring (int beginIndex) Returns a new string that is a substring of this string.
String	substring (int beginIndex, int endIndex) Returns a new string that is a substring of this string.
char[]	toCharArray () Converts this string to a new character array.
String	toLowerCase () Converts all of the characters in this String to lower case using the rules of the default locale.
String	toLowerCase (Locale locale) Converts all of the characters in this String to lower case using the rules of the given Locale .
String	toString () This object (which is already a string!) is itself returned.
String	toUpperCase () Converts all of the characters in this String to upper case using the rules of the default locale.
String	toUpperCase (Locale locale) Converts all of the characters in this String to upper case using the rules of the given Locale .
String	trim () Returns a copy of the string, with leading and trailing whitespace omitted.

static String	valueOf (boolean b) Returns the string representation of the <code>boolean</code> argument.
static String	valueOf (char c) Returns the string representation of the <code>char</code> argument.
static String	valueOf (char[] data) Returns the string representation of the <code>char</code> array argument.
static String	valueOf (char[] data, int offset, int count) Returns the string representation of a specific subarray of the <code>char</code> array argument.
static String	valueOf (double d) Returns the string representation of the <code>double</code> argument.
static String	valueOf (float f) Returns the string representation of the <code>float</code> argument.
static String	valueOf (int i) Returns the string representation of the <code>int</code> argument.
static String	valueOf (long l) Returns the string representation of the <code>long</code> argument.
static String	valueOf (Object obj) Returns the string representation of the <code>Object</code> argument.

Methods inherited from class [java.lang.Object](#)

[clone](#), [finalize](#), [getClass](#), [notify](#), [notifyAll](#), [wait](#), [wait](#), [wait](#)

Field Detail

CASE_INSENSITIVE_ORDER

```
public static final Comparator<String> CASE_INSENSITIVE_ORDER
```

A `Comparator` that orders `String` objects as by `compareToIgnoreCase`. This comparator is serializable.

Note that this `Comparator` does *not* take locale into account, and will result in an unsatisfactory ordering for certain locales. The `java.text` package provides *Collators* to allow locale-sensitive ordering.

Since:

1.2

See Also:[Collator.compare\(String, String\)](#)

Constructor Detail

String

```
public String()
```

Initializes a newly created `String` object so that it represents an empty character sequence. Note that use of this constructor is unnecessary since `Strings` are immutable.

String

```
public String(String original)
```

Initializes a newly created `String` object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string. Unless an explicit copy of `original` is needed, use of this constructor is unnecessary since `Strings` are immutable.

Parameters:

`original` - a `String`.

String

```
public String(char[] value)
```

Allocates a new `String` so that it represents the sequence of characters currently contained in the character array argument. The contents of the character array are copied; subsequent modification of the character array does not affect the newly created string.

Parameters:

`value` - the initial value of the string.

String

```
public String(char[] value,
               int offset,
               int count)
```

Allocates a new `String` that contains characters from a subarray of the character array argument. The `offset` argument is the index of the first character of the subarray and the `count` argument specifies the length of the subarray. The contents of the subarray are copied; subsequent modification of the character array does not affect the newly created string.

Parameters:

`value` - array that is the source of characters.
`offset` - the initial offset.
`count` - the length.

Throws:

[IndexOutOfBoundsException](#) - if the `offset` and `count` arguments index characters outside the bounds of the `value` array.

String

```
public String(int[] codePoints,
               int offset,
               int count)
```

Allocates a new `String` that contains characters from a subarray of the Unicode code point array argument. The `offset` argument is the index of the first code point of the subarray and the `count` argument specifies the length of the subarray. The contents of the subarray are converted to `chars`; subsequent modification of the `int` array does not affect the newly created string.

Parameters:

`codePoints` - array that is the source of Unicode code points.
`offset` - the initial offset.
`count` - the length.

Throws:

[IllegalArgumentException](#) - if any invalid Unicode code point is found in `codePoints`
[IndexOutOfBoundsException](#) - if the `offset` and `count` arguments index characters outside the bounds of the `codePoints` array.

Since:

1.5

String

```
public String(byte[] ascii,  
               int hibyte,  
               int offset,  
               int count)
```

Deprecated. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a charset name or that use the platform's default charset.*

Allocates a new `String` constructed from a subarray of an array of 8-bit integer values.

The `offset` argument is the index of the first byte of the subarray, and the `count` argument specifies the length of the subarray.

Each `byte` in the subarray is converted to a `char` as specified in the method above.

Parameters:

`ascii` - the bytes to be converted to characters.

`hibyte` - the top 8 bits of each 16-bit Unicode character.

`offset` - the initial offset.

`count` - the length.

Throws:

[IndexOutOfBoundsException](#) - if the `offset` or `count` argument is invalid.

See Also:

[String\(byte\[\], int\)](#), [String\(byte\[\], int, int, java.lang.String\)](#), [String\(byte\[\], int, int\)](#), [String\(byte\[\], java.lang.String\)](#), [String\(byte\[\]\)](#)

String

```
public String(byte[] ascii,  
               int hibyte)
```

Deprecated. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a charset name or that use the platform's default charset.*

Allocates a new `String` containing characters constructed from an array of 8-bit integer values. Each character `c` in the resulting string is constructed from the corresponding component `b` in the byte array such that:

$$c == (\text{char})(((\text{hibyte} \& 0\text{xff}) \ll 8) \mid (b \& 0\text{xff}))$$

Parameters:

`ascii` - the bytes to be converted to characters.

`hibyte` - the top 8 bits of each 16-bit Unicode character.

See Also:

[String\(byte\[\], int, int, java.lang.String\)](#), [String\(byte\[\], int, int\)](#), [String\(byte\[\], java.lang.String\)](#), [String\(byte\[\]\)](#)

String

```
public String(byte[] bytes,
               int offset,
               int length,
               String charsetName)
    throws UnsupportedEncodingException
```

Constructs a new `String` by decoding the specified subarray of bytes using the specified charset. The length of the new `String` is a function of the charset, and hence may not be equal to the length of the subarray.

The behavior of this constructor when the given bytes are not valid in the given charset is unspecified. The [CharsetDecoder](#) class should be used when more control over the decoding process is required.

Parameters:

`bytes` - the bytes to be decoded into characters

`offset` - the index of the first byte to decode

`length` - the number of bytes to decode

`charsetName` - the name of a supported [charset](#)

Throws:

[UnsupportedEncodingException](#) - if the named charset is not supported

[IndexOutOfBoundsException](#) - if the `offset` and `length` arguments index characters outside the bounds of the `bytes` array

Since:

JDK1.1

String

```
public String(byte[] bytes,  
              String charsetName)  
    throws UnsupportedEncodingException
```

Constructs a new `String` by decoding the specified array of bytes using the specified charset. The length of the new `String` is a function of the charset, and hence may not be equal to the length of the byte array.

The behavior of this constructor when the given bytes are not valid in the given charset is unspecified. The [CharsetDecoder](#) class should be used when more control over the decoding process is required.

Parameters:

`bytes` - the bytes to be decoded into characters

`charsetName` - the name of a supported [charset](#)

Throws:

[UnsupportedEncodingException](#) - If the named charset is not supported

Since:

JDK1.1

String

```
public String(byte[] bytes,  
              int offset,  
              int length)
```

Constructs a new `String` by decoding the specified subarray of bytes using the platform's default charset. The length of the new `String` is a function of the charset, and hence may not be equal to the length of the subarray.

The behavior of this constructor when the given bytes are not valid in the default charset is unspecified. The [CharsetDecoder](#) class should be used when more control over the decoding process is required.

Parameters:

`bytes` - the bytes to be decoded into characters

`offset` - the index of the first byte to decode

`length` - the number of bytes to decode

Throws:

[IndexOutOfBoundsException](#) - if the `offset` and the `length` arguments index characters outside the bounds of the `bytes` array

Since:

JDK1.1

String

```
public String(byte[] bytes)
```

Constructs a new `String` by decoding the specified array of bytes using the platform's default charset. The length of the new `String` is a function of the charset, and hence may not be equal to the length of the byte array.

The behavior of this constructor when the given bytes are not valid in the default charset is unspecified. The [CharsetDecoder](#) class should be used when more control over the decoding process is required.

Parameters:

`bytes` - the bytes to be decoded into characters

Since:

JDK1.1

String

```
public String(StringBuffer buffer)
```

Allocates a new string that contains the sequence of characters currently contained in the string buffer argument. The contents of the string buffer are copied; subsequent modification of the string buffer does not affect the newly created string.

Parameters:

`buffer` - a `StringBuffer`.

String

```
public String(StringBuilder builder)
```

Allocates a new string that contains the sequence of characters currently contained in the string builder argument. The contents of the string builder are copied; subsequent modification of the string builder does not affect the newly created string.

This constructor is provided to ease migration to `StringBuilder`. Obtaining a string from a string builder via the `toString` method is likely to run faster and is generally preferred.

Parameters:

builder - a `StringBuilder`

Since:

1.5

Method Detail

length

```
public int length()
```

Returns the length of this string. The length is equal to the number of 16-bit Unicode characters in the string.

Specified by:

[length](#) in interface [CharSequence](#)

Returns:

the length of the sequence of characters represented by this object.

charAt

```
public char charAt(int index)
```

Returns the `char` value at the specified index. An index ranges from 0 to `length() - 1`. The first `char` value of the sequence is at index 0, the next at index 1, and so on, as for array indexing.

If the `char` value specified by the index is a [surrogate](#), the surrogate value is returned.

Specified by:

[charAt](#) in interface [CharSequence](#)

Parameters:

`index` - the index of the `char` value.

Returns:

the `char` value at the specified index of this string. The first `char` value is at index `0`.

Throws:

[IndexOutOfBoundsException](#) - if the `index` argument is negative or not less than the length of this string.

codePointAt

```
public int codePointAt(int index)
```

Returns the character (Unicode code point) at the specified index. The index refers to `char` values (Unicode code units) and ranges from `0` to [length\(\)](#) - `1`.

If the `char` value specified at the given index is in the high-surrogate range, the following index is less than the length of this `String`, and the `char` value at the following index is in the low-surrogate range, then the supplementary code point corresponding to this surrogate pair is returned. Otherwise, the `char` value at the given index is returned.

Parameters:

`index` - the index to the `char` values

Returns:

the code point value of the character at the `index`

Throws:

[IndexOutOfBoundsException](#) - if the `index` argument is negative or not less than the length of this string.

Since:

1.5

codePointBefore

```
public int codePointBefore(int index)
```

Returns the character (Unicode code point) before the specified index. The index refers to `char` values (Unicode code units) and ranges from `1` to [length](#).

If the `char` value at (`index` - `1`) is in the low-surrogate range, (`index` - `2`) is not negative, and the `char` value at (`index` - `2`) is in the high-surrogate range, then the supplementary code point value of the surrogate pair is returned. If the `char` value at `index` - `1` is an unpaired low-surrogate or a high-surrogate, the surrogate value is returned.

Parameters:

`index` - the index following the code point that should be returned

Returns:

the Unicode code point value before the given index.

Throws:

[IndexOutOfBoundsException](#) - if the `index` argument is less than 1 or greater than the length of this string.

Since:

1.5

codePointCount

```
public int codePointCount(int beginIndex,
                        int endIndex)
```

Returns the number of Unicode code points in the specified text range of this `String`. The text range begins at the specified `beginIndex` and extends to the `char` at index `endIndex - 1`. Thus the length (in chars) of the text range is `endIndex - beginIndex`. Unpaired surrogates within the text range count as one code point each.

Parameters:

`beginIndex` - the index to the first `char` of the text range.

`endIndex` - the index after the last `char` of the text range.

Returns:

the number of Unicode code points in the specified text range

Throws:

[IndexOutOfBoundsException](#) - if the `beginIndex` is negative, or `endIndex` is larger than the length of this `String`, or `beginIndex` is larger than `endIndex`.

Since:

1.5

offsetByCodePoints

```
public int offsetByCodePoints(int index,
                            int codePointOffset)
```

Returns the index within this `String` that is offset from the given `index` by `codePointOffset` code points. Unpaired surrogates within the text range given by `index` and `codePointOffset` count as one code point each.

Parameters:

`index` - the index to be offset

`codePointOffset` - the offset in code points

Returns:

the index within this `String`

Throws:

[IndexOutOfBoundsException](#) - if `index` is negative or larger than the length of this `String`, or if `codePointOffset` is positive and the substring starting with `index` has fewer than `codePointOffset` code points, or if `codePointOffset` is negative and the substring before `index` has fewer than the absolute value of `codePointOffset` code points.

Since:

1.5

getChars

```
public void getChars(int srcBegin,
                    int srcEnd,
                    char[] dst,
                    int dstBegin)
```

Copies characters from this string into the destination character array.

The first character to be copied is at index `srcBegin`; the last character to be copied is at index `srcEnd-1` (thus the total number of characters to be copied is `srcEnd-srcBegin`). The characters are copied into the subarray of `dst` starting at index `dstBegin` and ending at index:

$$\text{dstbegin} + (\text{srcEnd} - \text{srcBegin}) - 1$$
Parameters:

`srcBegin` - index of the first character in the string to copy.

`srcEnd` - index after the last character in the string to copy.

`dst` - the destination array.

`dstBegin` - the start offset in the destination array.

Throws:

[IndexOutOfBoundsException](#) - If any of the following is true:

- `srcBegin` is negative.
- `srcBegin` is greater than `srcEnd`
- `srcEnd` is greater than the length of this string
- `dstBegin` is negative
- `dstBegin+(srcEnd-srcBegin)` is larger than `dst.length`

getBytes

```
public void getBytes(int srcBegin,  
                     int srcEnd,  
                     byte[] dst,  
                     int dstBegin)
```

Deprecated. *This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the `getBytes()` method, which uses the platform's default charset.*

Copies characters from this string into the destination byte array. Each byte receives the 8 low-order bits of the corresponding character. The eight high-order bits of each character are not copied and do not participate in the transfer in any way.

The first character to be copied is at index `srcBegin`; the last character to be copied is at index `srcEnd - 1`. The total number of characters to be copied is `srcEnd - srcBegin`. The characters, converted to bytes, are copied into the subarray of `dst` starting at index `dstBegin` and ending at index:

$$\text{dstbegin} + (\text{srcEnd} - \text{srcBegin}) - 1$$

Parameters:

`srcBegin` - index of the first character in the string to copy.

`srcEnd` - index after the last character in the string to copy.

`dst` - the destination array.

`dstBegin` - the start offset in the destination array.

Throws:

[IndexOutOfBoundsException](#) - if any of the following is true:

- `srcBegin` is negative
- `srcBegin` is greater than `srcEnd`
- `srcEnd` is greater than the length of this String
- `dstBegin` is negative
- `dstBegin + (srcEnd - srcBegin)` is larger than `dst.length`

getBytes

```
public byte[] getBytes(String charsetName)
```

throws [UnsupportedEncodingException](#)

Encodes this `String` into a sequence of bytes using the named charset, storing the result into a new byte array.

The behavior of this method when this string cannot be encoded in the given charset is unspecified. The [CharsetEncoder](#) class should be used when more control over the encoding process is required.

Parameters:

charsetName - the name of a supported [charset](#)

Returns:

The resultant byte array

Throws:

[UnsupportedEncodingException](#) - If the named charset is not supported

Since:

JDK1.1

getBytes

```
public byte[] getBytes()
```

Encodes this `String` into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

The behavior of this method when this string cannot be encoded in the default charset is unspecified. The [CharsetEncoder](#) class should be used when more control over the encoding process is required.

Returns:

The resultant byte array

Since:

JDK1.1

equals

```
public boolean equals(Object anObject)
```

Compares this string to the specified object. The result is `true` if and only if the argument is

not null and is a `String` object that represents the same sequence of characters as this object.

Overrides:

[equals](#) in class [Object](#)

Parameters:

`anObject` - the object to compare this `String` against.

Returns:

true if the `String` are equal; false otherwise.

See Also:

[compareTo\(java.lang.String\)](#), [equalsIgnoreCase\(java.lang.String\)](#)

contentEquals

```
public boolean contentEquals(StringBuffer sb)
```

Returns true if and only if this `String` represents the same sequence of characters as the specified `StringBuffer`.

Parameters:

`sb` - the `StringBuffer` to compare to.

Returns:

true if and only if this `String` represents the same sequence of characters as the specified `StringBuffer`, otherwise false.

Throws:

[NullPointerException](#) - if `sb` is null

Since:

1.4

contentEquals

```
public boolean contentEquals(CharSequence cs)
```

Returns true if and only if this `String` represents the same sequence of char values as the specified sequence.

Parameters:

`cs` - the sequence to compare to.

Returns:

true if and only if this `String` represents the same sequence of char values as the specified sequence, otherwise false.

Throws:

[NullPointerException](#) - if `cs` is null

Since:

1.5

equalsIgnoreCase

```
public boolean equalsIgnoreCase(String anotherString)
```

Compares this `String` to another `String`, ignoring case considerations. Two strings are considered equal ignoring case if they are of the same length, and corresponding characters in the two strings are equal ignoring case.

Two characters `c1` and `c2` are considered the same, ignoring case if at least one of the following is true:

- The two characters are the same (as compared by the `==` operator).
- Applying the method [Character.toUpperCase\(char\)](#) to each character produces the same result.
- Applying the method [Character.toLowerCase\(char\)](#) to each character produces the same result.

Parameters:

`anotherString` - the `String` to compare this `String` against.

Returns:

true if the argument is not null and the `Strings` are equal, ignoring case; false otherwise.

See Also:

[equals\(Object\)](#), [Character.toLowerCase\(char\)](#), [Character.toUpperCase\(char\)](#)

compareTo

```
public int compareTo(String anotherString)
```

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string. The

result is a negative integer if this `String` object lexicographically precedes the argument string. The result is a positive integer if this `String` object lexicographically follows the argument string. The result is zero if the strings are equal; `compareTo` returns 0 exactly when the [equals\(Object\)](#) method would return `true`.

This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let k be the smallest such index; then the string whose character at position k has the smaller value, as determined by using the `<` operator, lexicographically precedes the other string. In this case, `compareTo` returns the difference of the two character values at position k in the two string -- that is, the value:

$$\text{this.charAt}(k) - \text{anotherString.charAt}(k)$$

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, `compareTo` returns the difference of the lengths of the strings -- that is, the value:

$$\text{this.length}() - \text{anotherString.length}()$$

Specified by:

[compareTo](#) in interface [Comparable<String>](#)

Parameters:

`anotherString` - the `String` to be compared.

Returns:

the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

compareToIgnoreCase

```
public int compareToIgnoreCase(String str)
```

Compares two strings lexicographically, ignoring case differences. This method returns an integer whose sign is that of calling `compareTo` with normalized versions of the strings where case differences have been eliminated by calling `Character.toLowerCase(Character.toUpperCase(character))` on each character.

Note that this method does *not* take locale into account, and will result in an unsatisfactory ordering for certain locales. The `java.text` package provides *collators* to allow locale-sensitive

ordering.

Parameters:

`str` - the `String` to be compared.

Returns:

a negative integer, zero, or a positive integer as the specified `String` is greater than, equal to, or less than this `String`, ignoring case considerations.

Since:

1.2

See Also:

[Collator.compare\(String, String\)](#)

regionMatches

```
public boolean regionMatches(int toffset,
                               String other,
                               int ooffset,
                               int len)
```

Tests if two string regions are equal.

A substring of this `String` object is compared to a substring of the argument `other`. The result is `true` if these substrings represent identical character sequences. The substring of this `String` object to be compared begins at index `toffset` and has length `len`. The substring of `other` to be compared begins at index `ooffset` and has length `len`. The result is `false` if and only if at least one of the following is true:

- `toffset` is negative.
- `ooffset` is negative.
- `toffset+len` is greater than the length of this `String` object.
- `ooffset+len` is greater than the length of the other argument.
- There is some nonnegative integer `k` less than `len` such that: `this.charAt(toffset+k) != other.charAt(ooffset+k)`

Parameters:

`toffset` - the starting offset of the subregion in this string.

`other` - the string argument.

`ooffset` - the starting offset of the subregion in the string argument.

`len` - the number of characters to compare.

Returns:

`true` if the specified subregion of this string exactly matches the specified subregion of the string argument; `false` otherwise.

regionMatches

```
public boolean regionMatches(boolean ignoreCase,  
                             int toffset,  
                             String other,  
                             int ooffset,  
                             int len)
```

Tests if two string regions are equal.

A substring of this `String` object is compared to a substring of the argument `other`. The result is `true` if these substrings represent character sequences that are the same, ignoring case if and only if `ignoreCase` is true. The substring of this `String` object to be compared begins at index `toffset` and has length `len`. The substring of `other` to be compared begins at index `ooffset` and has length `len`. The result is `false` if and only if at least one of the following is true:

- `toffset` is negative.
- `ooffset` is negative.
- `toffset+len` is greater than the length of this `String` object.
- `ooffset+len` is greater than the length of the other argument.
- `ignoreCase` is `false` and there is some nonnegative integer k less than `len` such that:

```
    this.charAt(toffset+k) != other.charAt(ooffset  
+k)
```

- `ignoreCase` is `true` and there is some nonnegative integer k less than `len` such that:

```
    Character.toLowerCase(this.charAt(toffset+k)) !=  
        Character.toLowerCase(other.charAt  
(ooffset+k))
```

and:

```
    Character.toUpperCase(this.charAt(toffset+k)) !=  
        Character.toUpperCase(other.charAt  
(ooffset+k))
```

Parameters:

`ignoreCase` - if `true`, ignore case when comparing characters.
`toffset` - the starting offset of the subregion in this string.
`other` - the string argument.
`ooffset` - the starting offset of the subregion in the string argument.
`len` - the number of characters to compare.

Returns:

`true` if the specified subregion of this string matches the specified subregion of the string argument; `false` otherwise. Whether the matching is exact or case insensitive depends on the `ignoreCase` argument.

startsWith

```
public boolean startsWith(String prefix,
                           int toffset)
```

Tests if this string starts with the specified prefix beginning a specified index.

Parameters:

`prefix` - the prefix.
`toffset` - where to begin looking in the string.

Returns:

`true` if the character sequence represented by the argument is a prefix of the substring of this object starting at index `toffset`; `false` otherwise. The result is `false` if `toffset` is negative or greater than the length of this `String` object; otherwise the result is the same as the result of the expression

```
this.substring(toffset).startsWith(prefix)
```

startsWith

```
public boolean startsWith(String prefix)
```

Tests if this string starts with the specified prefix.

Parameters:

`prefix` - the prefix.

Returns:

`true` if the character sequence represented by the argument is a prefix of the character

sequence represented by this string; `false` otherwise. Note also that `true` will be returned if the argument is an empty string or is equal to this `String` object as determined by the [equals\(Object\)](#) method.

Since:

1.0

endsWith

```
public boolean endsWith(String suffix)
```

Tests if this string ends with the specified suffix.

Parameters:

`suffix` - the suffix.

Returns:

`true` if the character sequence represented by the argument is a suffix of the character sequence represented by this object; `false` otherwise. Note that the result will be `true` if the argument is the empty string or is equal to this `String` object as determined by the [equals\(Object\)](#) method.

hashCode

```
public int hashCode()
```

Returns a hash code for this string. The hash code for a `String` object is computed as

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \dots + s[n-1]$$

using `int` arithmetic, where `s[i]` is the *i*th character of the string, *n* is the length of the string, and [^] indicates exponentiation. (The hash value of the empty string is zero.)

Overrides:

[hashCode](#) in class [Object](#)

Returns:

a hash code value for this object.

See Also:

[Object.equals\(java.lang.Object\)](#), [Hashtable](#)

indexOf

```
public int indexOf(int ch)
```

Returns the index within this string of the first occurrence of the specified character. If a character with value `ch` occurs in the character sequence represented by this `String` object, then the index (in Unicode code units) of the first such occurrence is returned. For values of `ch` in the range from 0 to 0xFFFF (inclusive), this is the smallest value k such that:

$$\text{this.charAt}(k) == \text{ch}$$

is true. For other values of `ch`, it is the smallest value k such that:

$$\text{this.codePointAt}(k) == \text{ch}$$

is true. In either case, if no such character occurs in this string, then `-1` is returned.

Parameters:

`ch` - a character (Unicode code point).

Returns:

the index of the first occurrence of the character in the character sequence represented by this object, or `-1` if the character does not occur.

indexOf

```
public int indexOf(int ch,  
                  int fromIndex)
```

Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

If a character with value `ch` occurs in the character sequence represented by this `String` object at an index no smaller than `fromIndex`, then the index of the first such occurrence is returned. For values of `ch` in the range from 0 to 0xFFFF (inclusive), this is the smallest value k such that:

$$(\text{this.charAt}(k) == \text{ch}) \ \&\& \ (k \geq \text{fromIndex})$$

is true. For other values of `ch`, it is the smallest value k such that:

$$(this.codePointAt(k) == ch) \ \&\& \ (k \geq fromIndex)$$

is true. In either case, if no such character occurs in this string at or after position `fromIndex`, then `-1` is returned.

There is no restriction on the value of `fromIndex`. If it is negative, it has the same effect as if it were zero: this entire string may be searched. If it is greater than the length of this string, it has the same effect as if it were equal to the length of this string: `-1` is returned.

All indices are specified in `char` values (Unicode code units).

Parameters:

`ch` - a character (Unicode code point).

`fromIndex` - the index to start the search from.

Returns:

the index of the first occurrence of the character in the character sequence represented by this object that is greater than or equal to `fromIndex`, or `-1` if the character does not occur.

lastIndexOf

```
public int lastIndexOf(int ch)
```

Returns the index within this string of the last occurrence of the specified character. For values of `ch` in the range from 0 to 0xFFFF (inclusive), the index (in Unicode code units) returned is the largest value k such that:

$$this.charAt(k) == ch$$

is true. For other values of `ch`, it is the largest value k such that:

$$this.codePointAt(k) == ch$$

is true. In either case, if no such character occurs in this string, then `-1` is returned. The `String` is searched backwards starting at the last character.

Parameters:

`ch` - a character (Unicode code point).

Returns:

the index of the last occurrence of the character in the character sequence represented by this object, or **-1** if the character does not occur.

lastIndexOf

```
public int lastIndexOf(int ch,
                       int fromIndex)
```

Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index. For values of `ch` in the range from 0 to 0xFFFF (inclusive), the index returned is the largest value *k* such that:

$$(this.charAt(k) == ch) \ \&\& \ (k \leq fromIndex)$$

is true. For other values of `ch`, it is the largest value *k* such that:

$$(this.codePointAt(k) == ch) \ \&\& \ (k \leq fromIndex)$$

is true. In either case, if no such character occurs in this string at or before position `fromIndex`, then **-1** is returned.

All indices are specified in `char` values (Unicode code units).

Parameters:

`ch` - a character (Unicode code point).

`fromIndex` - the index to start the search from. There is no restriction on the value of `fromIndex`. If it is greater than or equal to the length of this string, it has the same effect as if it were equal to one less than the length of this string: this entire string may be searched. If it is negative, it has the same effect as if it were **-1**: **-1** is returned.

Returns:

the index of the last occurrence of the character in the character sequence represented by this object that is less than or equal to `fromIndex`, or **-1** if the character does not occur before that point.

indexOf

```
public int indexOf(String str)
```

Returns the index within this string of the first occurrence of the specified substring. The integer returned is the smallest value k such that:

```
this.startsWith(str, k)
```

is true.

Parameters:

`str` - any string.

Returns:

if the string argument occurs as a substring within this object, then the index of the first character of the first such substring is returned; if it does not occur as a substring, `-1` is returned.

indexOf

```
public int indexOf(String str,
                  int fromIndex)
```

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index. The integer returned is the smallest value k for which:

```
k >= Math.min(fromIndex, str.length()) && this.
startsWith(str, k)
```

If no such value of k exists, then `-1` is returned.

Parameters:

`str` - the substring for which to search.

`fromIndex` - the index from which to start the search.

Returns:

the index within this string of the first occurrence of the specified substring, starting at the specified index.

lastIndexOf

```
public int lastIndexOf(String str)
```

Returns the index within this string of the rightmost occurrence of the specified substring. The rightmost empty string "" is considered to occur at the index value `this.length()`. The returned index is the largest value k such that

```
this.startsWith(str, k)
```

is true.

Parameters:

`str` - the substring to search for.

Returns:

if the string argument occurs one or more times as a substring within this object, then the index of the first character of the last such substring is returned. If it does not occur as a substring, `-1` is returned.

lastIndexOf

```
public int lastIndexOf(String str,
                      int fromIndex)
```

Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index. The integer returned is the largest value k such that:

```
k <= Math.min(fromIndex, str.length()) && this.
startsWith(str, k)
```

If no such value of k exists, then `-1` is returned.

Parameters:

`str` - the substring to search for.

`fromIndex` - the index to start the search from.

Returns:

the index within this string of the last occurrence of the specified substring.

substring

```
public String substring(int beginIndex)
```


Returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

Examples:

```
"unhappy".substring(2) returns "happy"  
"Harbison".substring(3) returns "bison"  
"emptiness".substring(9) returns "" (an empty string)
```

Parameters:

`beginIndex` - the beginning index, inclusive.

Returns:

the specified substring.

Throws:

[IndexOutOfBoundsException](#) - if `beginIndex` is negative or larger than the length of this `String` object.

substring

```
public String substring(int beginIndex,  
                        int endIndex)
```

Returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex` - 1. Thus the length of the substring is `endIndex - beginIndex`.

Examples:

```
"hamburger".substring(4, 8) returns "urge"  
"smiles".substring(1, 5) returns "mile"
```

Parameters:

`beginIndex` - the beginning index, inclusive.

`endIndex` - the ending index, exclusive.

Returns:

the specified substring.

Throws:

[IndexOutOfBoundsException](#) - if the `beginIndex` is negative, or `endIndex` is larger than the length of this `String` object, or `beginIndex` is larger than `endIndex`.

subSequence

```
public CharSequence subSequence(int beginIndex,  
                                   int endIndex)
```

Returns a new character sequence that is a subsequence of this sequence.

An invocation of this method of the form

```
    str.subSequence(begin, end)
```

behaves in exactly the same way as the invocation

```
    str.substring(begin, end)
```

This method is defined so that the `String` class can implement the [CharSequence](#) interface.

Specified by:

[subSequence](#) in interface [CharSequence](#)

Parameters:

`beginIndex` - the begin index, inclusive.

`endIndex` - the end index, exclusive.

Returns:

the specified subsequence.

Throws:

[IndexOutOfBoundsException](#) - if `beginIndex` or `endIndex` are negative, if `endIndex` is greater than `length()`, or if `beginIndex` is greater than `startIndex`

Since:

1.4

concat

```
public String concat(String str)
```

Concatenates the specified string to the end of this string.

If the length of the argument string is 0, then this `String` object is returned. Otherwise, a new `String` object is created, representing a character sequence that is the concatenation of

the character sequence represented by this `String` object and the character sequence represented by the argument string.

Examples:

```
"cares".concat("s") returns "caress"  
"to".concat("get").concat("her") returns "together"
```

Parameters:

`str` - the `String` that is concatenated to the end of this `String`.

Returns:

a string that represents the concatenation of this object's characters followed by the string argument's characters.

replace

```
public String replace(char oldChar,  
                     char newChar)
```

Returns a new string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.

If the character `oldChar` does not occur in the character sequence represented by this `String` object, then a reference to this `String` object is returned. Otherwise, a new `String` object is created that represents a character sequence identical to the character sequence represented by this `String` object, except that every occurrence of `oldChar` is replaced by an occurrence of `newChar`.

Examples:

```
"mesquite in your cellar".replace('e', 'o')  
    returns "mosquito in your collar"  
"the war of baronets".replace('r', 'y')  
    returns "the way of bayonets"  
"sparring with a purple porpoise".replace('p', 't')  
    returns "starring with a turtle tortoise"  
"JonL".replace('q', 'x') returns "JonL" (no change)
```

Parameters:

`oldChar` - the old character.

`newChar` - the new character.

Returns:

a string derived from this string by replacing every occurrence of `oldChar` with `newChar`.

matches

```
public boolean matches(String regex)
```

Tells whether or not this string matches the given [regular expression](#).

An invocation of this method of the form `str.matches(regex)` yields exactly the same result as the expression

[Pattern.matches\(regex, str\)](#)

Parameters:

`regex` - the regular expression to which this string is to be matched

Returns:

`true` if, and only if, this string matches the given regular expression

Throws:

[PatternSyntaxException](#) - if the regular expression's syntax is invalid

Since:

1.4

See Also:

[Pattern](#)

contains

```
public boolean contains(CharSequence s)
```

Returns `true` if and only if this string contains the specified sequence of char values.

Parameters:

`s` - the sequence to search for

Returns:

`true` if this string contains `s`, `false` otherwise

Throws:

[NullPointerException](#) - if `s` is `null`

Since:

1.5

replaceFirst

```
public String replaceFirst(String regex,  
                           String replacement)
```

Replaces the first substring of this string that matches the given [regular expression](#) with the given replacement.

An invocation of this method of the form `str.replaceFirst(regex, repl)` yields exactly the same result as the expression

```
Pattern.compile(regex).matcher(str).replaceFirst(repl)
```

Parameters:

regex - the regular expression to which this string is to be matched

Returns:

The resulting `String`

Throws:

[PatternSyntaxException](#) - if the regular expression's syntax is invalid

Since:

1.4

See Also:

[Pattern](#)

replaceAll

```
public String replaceAll(String regex,  
                         String replacement)
```

Replaces each substring of this string that matches the given [regular expression](#) with the given replacement.

An invocation of this method of the form `str.replaceAll(regex, repl)` yields exactly the same result as the expression

```
Pattern.compile(regex).matcher(str).replaceAll(repl)
```

Parameters:

`regex` - the regular expression to which this string is to be matched

Returns:

The resulting String

Throws:

[PatternSyntaxException](#) - if the regular expression's syntax is invalid

Since:

1.4

See Also:

[Pattern](#)

replace

```
public String replace(CharSequence target,  
                       CharSequence replacement)
```

Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence. The replacement proceeds from the beginning of the string to the end, for example, replacing "aa" with "b" in the string "aaa" will result in "ba" rather than "ab".

Parameters:

`target` - The sequence of char values to be replaced

`replacement` - The replacement sequence of char values

Returns:

The resulting string

Throws:

[NullPointerException](#) - if `target` or `replacement` is null.

Since:

1.5

split

```
public String[] split(String regex,  
                      int limit)
```

Splits this string around matches of the given regular expression.

The array returned by this method contains each substring of this string that is terminated by another substring that matches the given expression or is terminated by the end of the string.

The substrings in the array are in the order in which they occur in this string. If the expression does not match any part of the input then the resulting array has just one element, namely this string.

The `limit` parameter controls the number of times the pattern is applied and therefore affects the length of the resulting array. If the limit n is greater than zero then the pattern will be applied at most $n - 1$ times, the array's length will be no greater than n , and the array's last entry will contain all input beyond the last matched delimiter. If n is non-positive then the pattern will be applied as many times as possible and the array can have any length. If n is zero then the pattern will be applied as many times as possible, the array can have any length, and trailing empty strings will be discarded.

The string `"boo:and:foo"`, for example, yields the following results with these parameters:

Regex	Limit	Result
:	2	{ "boo", "and:foo" }
:	5	{ "boo", "and", "foo" }
:	-2	{ "boo", "and", "foo" }
o	5	{ "b", "", ":and:f", "", "" }
o	-2	{ "b", "", ":and:f", "", "" }
o	0	{ "b", "", ":and:f" }

An invocation of this method of the form `str.split(regex, n)` yields the same result as the expression

```
Pattern.compile\(regex\).split\(str, n\)
```

Parameters:

`regex` - the delimiting regular expression

`limit` - the result threshold, as described above

Returns:

the array of strings computed by splitting this string around matches of the given regular expression

Throws:

[PatternSyntaxException](#) - if the regular expression's syntax is invalid

Since:

1.4

See Also:

[Pattern](#)

```
public String[] split(String regex)
```

Splits this string around matches of the given regular expression.

This method works as if by invoking the two-argument [split](#) method with the given expression and a limit argument of zero. Trailing empty strings are therefore not included in the resulting array.

The string "boo:and:foo", for example, yields the following results with these expressions:

Regex	Result
:	{ "boo", "and", "foo" }
o	{ "b", "", ":and:f" }

Parameters:

regex - the delimiting regular expression

Returns:

the array of strings computed by splitting this string around matches of the given regular expression

Throws:

[PatternSyntaxException](#) - if the regular expression's syntax is invalid

Since:

1.4

See Also:

[Pattern](#)

toLowerCase

```
public String toLowerCase(Locale locale)
```

Converts all of the characters in this [String](#) to lower case using the rules of the given [Locale](#). Case mapping is based on the Unicode Standard version specified by the [Character](#) class. Since case mappings are not always 1:1 char mappings, the resulting [String](#) may be a different length than the original [String](#).

Examples of lowercase mappings are in the following table:

Language Code of Locale	Upper Case	Lower Case	Description

tr (Turkish)	\u0130	\u0069	capital letter I with dot above -> small letter i
tr (Turkish)	\u0049	\u0131	capital letter I -> small letter dotless i
(all)	French Fries	french fries	lowercased all chars in String
(all)	IX ΘY Σ	ix θυ ς	lowercased all chars in String

Parameters:

`locale` - use the case transformation rules for this locale

Returns:

the `String`, converted to lowercase.

Since:

1.1

See Also:

[toLowerCase\(\)](#), [toUpperCase\(\)](#), [toUpperCase\(Locale\)](#)

toLowerCase

```
public String toLowerCase()
```

Converts all of the characters in this `String` to lower case using the rules of the default locale. This is equivalent to calling `toLowerCase(Locale.getDefault())`.

Returns:

the `String`, converted to lowercase.

See Also:

[toLowerCase\(Locale\)](#)

toUpperCase

```
public String toUpperCase(Locale locale)
```

Converts all of the characters in this `String` to upper case using the rules of the given `Locale`. Case mapping is based on the Unicode Standard version specified by the [Character](#) class. Since case mappings are not always 1:1 char mappings, the resulting `String` may be a different length than the original `String`.

Examples of locale-sensitive and 1:M case mappings are in the following table.

Language Code of Locale	Lower Case	Upper Case	Description
tr (Turkish)	\u0069	\u0130	small letter i -> capital letter I with dot above
tr (Turkish)	\u0131	\u0049	small letter dotless i -> capital letter I
(all)	\u00df	\u0053 \u0053	small letter sharp s -> two letters: SS
(all)	Fahrvergnügen	FAHRVERGNÜGEN	

Parameters:

`locale` - use the case transformation rules for this locale

Returns:

the `String`, converted to uppercase.

Since:

1.1

See Also:

[toUpperCase\(\)](#), [toLowerCase\(\)](#), [toLowerCase\(Locale\)](#)

toUpperCase

```
public String toUpperCase()
```

Converts all of the characters in this `String` to upper case using the rules of the default locale. This method is equivalent to `toUpperCase(Locale.getDefault())`.

Returns:

the `String`, converted to uppercase.

See Also:

[toUpperCase\(Locale\)](#)

trim

```
public String trim()
```

Returns a copy of the string, with leading and trailing whitespace omitted.

If this `String` object represents an empty character sequence, or the first and last characters

of character sequence represented by this `String` object both have codes greater than `'\u0020'` (the space character), then a reference to this `String` object is returned.

Otherwise, if there is no character with a code greater than `'\u0020'` in the string, then a new `String` object representing an empty string is created and returned.

Otherwise, let k be the index of the first character in the string whose code is greater than `'\u0020'`, and let m be the index of the last character in the string whose code is greater than `'\u0020'`. A new `String` object is created, representing the substring of this string that begins with the character at index k and ends with the character at index m -that is, the result of `this.substring( $k$ ,  $m+1$ )`.

This method may be used to trim [whitespace](#) from the beginning and end of a string; in fact, it trims all ASCII control characters as well.

Returns:

A copy of this string with leading and trailing white space removed, or this string if it has no leading or trailing white space.

toString

```
public String toString()
```

This object (which is already a string!) is itself returned.

Specified by:

[toString](#) in interface [CharSequence](#)

Overrides:

[toString](#) in class [Object](#)

Returns:

the string itself.

toCharArray

```
public char[] toCharArray()
```

Converts this string to a new character array.

Returns:

a newly allocated character array whose length is the length of this string and whose

contents are initialized to contain the character sequence represented by this string.

format

```
public static String format(String format,  
                           Object... args)
```

Returns a formatted string using the specified format string and arguments.

The locale always used is the one returned by [Locale.getDefault\(\)](#).

Parameters:

format - A [format string](#)

args - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The number of arguments is variable and may be zero. The maximum number of arguments is limited by the maximum dimension of a Java array as defined by the [Java Virtual Machine Specification](#). The behaviour on a `null` argument depends on the [conversion](#).

Returns:

A formatted string

Throws:

[IllegalFormatException](#) - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the [Details](#) section of the formatter class specification.

[NullPointerException](#) - If the format is `null`

Since:

1.5

See Also:

[Formatter](#)

format

```
public static String format(Locale l,  
                           String format,  
                           Object... args)
```

Returns a formatted string using the specified locale, format string, and arguments.

Parameters:

`l` - The [locale](#) to apply during formatting. If `l` is `null` then no localization is applied.

`format` - A [format string](#)

`args` - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The number of arguments is variable and may be zero. The maximum number of arguments is limited by the maximum dimension of a Java array as defined by the [Java Virtual Machine Specification](#). The behaviour on a `null` argument depends on the [conversion](#).

Returns:

A formatted string

Throws:

`IllegalArgumentException` - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the [Details](#) section of the formatter class specification

[NullPointerException](#) - If the format is `null`

Since:

1.5

See Also:

[Formatter](#)

valueOf

```
public static String valueOf(Object obj)
```

Returns the string representation of the `Object` argument.

Parameters:

`obj` - an `Object`.

Returns:

if the argument is `null`, then a string equal to `"null"`; otherwise, the value of `obj.toString()` is returned.

See Also:

[Object.toString\(\)](#)

valueOf

```
public static String valueOf(char[] data)
```

Returns the string representation of the `char` array argument. The contents of the character array are copied; subsequent modification of the character array does not affect the newly created string.

Parameters:

`data` - a `char` array.

Returns:

a newly allocated string representing the same sequence of characters contained in the character array argument.

valueOf

```
public static String valueOf(char[] data,  
                             int offset,  
                             int count)
```

Returns the string representation of a specific subarray of the `char` array argument.

The `offset` argument is the index of the first character of the subarray. The `count` argument specifies the length of the subarray. The contents of the subarray are copied; subsequent modification of the character array does not affect the newly created string.

Parameters:

`data` - the character array.

`offset` - the initial offset into the value of the `String`.

`count` - the length of the value of the `String`.

Returns:

a string representing the sequence of characters contained in the subarray of the character array argument.

Throws:

[IndexOutOfBoundsException](#) - if `offset` is negative, or `count` is negative, or `offset+count` is larger than `data.length`.

copyValueOf

```
public static String copyValueOf(char[] data,  
                                 int offset,  
                                 int count)
```

Returns a `String` that represents the character sequence in the array specified.

Parameters:

`data` - the character array.
`offset` - initial offset of the subarray.
`count` - length of the subarray.

Returns:

a `String` that contains the characters of the specified subarray of the character array.

copyValueOf

```
public static String copyValueOf(char[] data)
```

Returns a `String` that represents the character sequence in the array specified.

Parameters:

`data` - the character array.

Returns:

a `String` that contains the characters of the character array.

valueOf

```
public static String valueOf(boolean b)
```

Returns the string representation of the `boolean` argument.

Parameters:

`b` - a `boolean`.

Returns:

if the argument is `true`, a string equal to `"true"` is returned; otherwise, a string equal to `"false"` is returned.

valueOf

```
public static String valueOf(char c)
```

Returns the string representation of the `char` argument.

Parameters:

`c` - a `char`.

Returns:

a string of length `1` containing as its single character the argument `c`.

valueOf

```
public static String valueOf(int i)
```

Returns the string representation of the `int` argument.

The representation is exactly the one returned by the `Integer.toString` method of one argument.

Parameters:

`i` - an `int`.

Returns:

a string representation of the `int` argument.

See Also:

[Integer.toString\(int, int\)](#)

valueOf

```
public static String valueOf(long l)
```

Returns the string representation of the `long` argument.

The representation is exactly the one returned by the `Long.toString` method of one argument.

Parameters:

`l` - a `long`.

Returns:

a string representation of the `long` argument.

See Also:

[Long.toString\(long\)](#)

valueOf


```
public static String valueOf(float f)
```

Returns the string representation of the `float` argument.

The representation is exactly the one returned by the `Float.toString` method of one argument.

Parameters:

`f` - a `float`.

Returns:

a string representation of the `float` argument.

See Also:

[Float.toString\(float\)](#)

valueOf

```
public static String valueOf(double d)
```

Returns the string representation of the `double` argument.

The representation is exactly the one returned by the `Double.toString` method of one argument.

Parameters:

`d` - a `double`.

Returns:

a string representation of the `double` argument.

See Also:

[Double.toString\(double\)](#)

intern

```
public String intern()
```

Returns a canonical representation for the string object.

A pool of strings, initially empty, is maintained privately by the class `String`.

When the `intern` method is invoked, if the pool already contains a string equal to this `String`

object as determined by the [equals\(Object\)](#) method, then the string from the pool is returned. Otherwise, this `String` object is added to the pool and a reference to this `String` object is returned.

It follows that for any two strings `s` and `t`, `s.intern() == t.intern()` is `true` if and only if `s.equals(t)` is `true`.

All literal strings and string-valued constant expressions are interned. String literals are defined in §3.10.5 of the [Java Language Specification](#)

Returns:

a string that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*Java™ 2 Platform
Std. Ed. v1.5.0*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)*Java™ 2 Platform
Std. Ed. v1.5.0*[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#) [All Classes](#)SUMMARY: NESTED | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: FIELD | [CONSTR](#) | [METHOD](#)

java.io

Class PrintStream

[java.lang.Object](#)└ [java.io.OutputStream](#)└ [java.io.FilterOutputStream](#)└ **java.io.PrintStream**

All Implemented Interfaces:

[Closeable](#), [Flushable](#), [Appendable](#)

Direct Known Subclasses:

[LogStream](#)

```
public class PrintStream
```

extends [FilterOutputStream](#)implements [Appendable](#), [Closeable](#)

A `PrintStream` adds functionality to another output stream, namely the ability to print representations of various data values conveniently. Two other features are provided as well. Unlike other output streams, a `PrintStream` never throws an `IOException`; instead, exceptional situations merely set an internal flag that can be tested via the `checkError` method. Optionally, a `PrintStream` can be created so as to flush automatically; this means that the `flush` method is automatically invoked after a byte array is written, one of the `println` methods is invoked, or a newline character or byte (`'\n'`) is written.

All characters printed by a `PrintStream` are converted into bytes using the platform's default character encoding. The [PrintWriter](#) class should be used in situations that require writing characters rather than bytes.

Since:

JDK1.0

Field Summary

Fields inherited from class java.io.[FilterOutputStream](#)

[out](#)

Constructor Summary

[PrintStream](#)([File](#) file)

Creates a new print stream, without automatic line flushing, with the specified file.

[PrintStream](#)([File](#) file, [String](#) csn)

Creates a new print stream, without automatic line flushing, with the specified file and charset.

[PrintStream](#)([OutputStream](#) out)

Create a new print stream.

[PrintStream](#)([OutputStream](#) out, boolean autoFlush)

Create a new print stream.

[PrintStream](#)([OutputStream](#) out, boolean autoFlush, [String](#) encoding)

Create a new print stream.

[PrintStream](#)([String](#) fileName)

Creates a new print stream, without automatic line flushing, with the specified file name.

[PrintStream](#)([String](#) fileName, [String](#) csn)

Creates a new print stream, without automatic line flushing, with the specified file name and charset.

Method Summary

[PrintStream](#) [append](#)(char c)

Appends the specified character to this output stream.

[PrintStream](#) [append](#)([CharSequence](#) csq)

Appends the specified character sequence to this output stream.

boolean [checkError](#)()

Flush the stream and check its error state.

void [close](#)()

Close the stream.

void [flush](#)()

Flush the stream.

PrintStream	format (Locale l, String format, Object ... args) Writes a formatted string to this output stream using the specified format string and arguments.
PrintStream	format (String format, Object ... args) Writes a formatted string to this output stream using the specified format string and arguments.
void	print (boolean b) Print a boolean value.
void	print (char c) Print a character.
void	print (char[] s) Print an array of characters.
void	print (double d) Print a double-precision floating-point number.
void	print (float f) Print a floating-point number.
void	print (int i) Print an integer.
void	print (long l) Print a long integer.
void	print (Object obj) Print an object.
void	print (String s) Print a string.
PrintStream	printf (Locale l, String format, Object ... args) A convenience method to write a formatted string to this output stream using the specified format string and arguments.
PrintStream	printf (String format, Object ... args) A convenience method to write a formatted string to this output stream using the specified format string and arguments.
void	println () Terminate the current line by writing the line separator string.
void	println (boolean x) Print a boolean and then terminate the line.
void	println (char x) Print a character and then terminate the line.

void	println (char[] x) Print an array of characters and then terminate the line.
void	println (double x) Print a double and then terminate the line.
void	println (float x) Print a float and then terminate the line.
void	println (int x) Print an integer and then terminate the line.
void	println (long x) Print a long and then terminate the line.
void	println (Object x) Print an Object and then terminate the line.
void	println (String x) Print a String and then terminate the line.
protected void	setError () Set the error state of the stream to true.
void	write (byte[] buf, int off, int len) Write len bytes from the specified byte array starting at offset off to this stream.
void	write (int b) Write the specified byte to this stream.

Methods inherited from class java.io.[FilterOutputStream](#)

[write](#)

Methods inherited from class java.lang.[Object](#)

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

Constructor Detail

PrintStream

```
public PrintStream(OutputStream out)
```

Create a new print stream. This stream will not flush automatically.

Parameters:

out - The output stream to which values and objects will be printed

See Also:

[PrintWriter.PrintWriter\(java.io.OutputStream\)](#)

PrintStream

```
public PrintStream(OutputStream out,  
                   boolean autoFlush)
```

Create a new print stream.

Parameters:

out - The output stream to which values and objects will be printed

autoFlush - A boolean; if true, the output buffer will be flushed whenever a byte array is written, one of the `println` methods is invoked, or a newline character or byte (`'\n'`) is written

See Also:

[PrintWriter.PrintWriter\(java.io.OutputStream, boolean\)](#)

PrintStream

```
public PrintStream(OutputStream out,  
                   boolean autoFlush,  
                   String encoding)  
    throws UnsupportedEncodingException
```

Create a new print stream.

Parameters:

out - The output stream to which values and objects will be printed

autoFlush - A boolean; if true, the output buffer will be flushed whenever a byte array is written, one of the `println` methods is invoked, or a newline character or byte (`'\n'`) is written

encoding - The name of a supported [character encoding](#)

Throws:

[UnsupportedEncodingException](#) - If the named encoding is not supported

PrintStream

```
public PrintStream(String fileName)
    throws FileNotFoundException
```

Creates a new print stream, without automatic line flushing, with the specified file name. This convenience constructor creates the necessary intermediate [OutputStreamWriter](#), which will encode characters using the [default charset](#) for this instance of the Java virtual machine.

Parameters:

`fileName` - The name of the file to use as the destination of this print stream. If the file exists, then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.

Throws:

[FileNotFoundException](#) - If the given file object does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file

[SecurityException](#) - If a security manager is present and [checkWrite\(fileName\)](#) denies write access to the file

Since:

1.5

PrintStream

```
public PrintStream(String fileName,
    String csn)
    throws FileNotFoundException,
    UnsupportedEncodingException
```

Creates a new print stream, without automatic line flushing, with the specified file name and charset. This convenience constructor creates the necessary intermediate [OutputStreamWriter](#), which will encode characters using the provided charset.

Parameters:

`fileName` - The name of the file to use as the destination of this print stream. If the file exists, then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.

`csn` - The name of a supported [charset](#)

Throws:

[FileNotFoundException](#) - If the given file object does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file

[SecurityException](#) - If a security manager is present and [checkWrite\(fileName\)](#) denies write access to the file

[UnsupportedEncodingException](#) - If the named charset is not supported

Since:

1.5

PrintStream

```
public PrintStream(File file)
    throws FileNotFoundException
```

Creates a new print stream, without automatic line flushing, with the specified file. This convenience constructor creates the necessary intermediate [OutputStreamWriter](#), which will encode characters using the [default charset](#) for this instance of the Java virtual machine.

Parameters:

`file` - The file to use as the destination of this print stream. If the file exists, then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.

Throws:

[FileNotFoundException](#) - If the given file object does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file

[SecurityException](#) - If a security manager is present and [checkWrite\(file.getPath\(\)\)](#) denies write access to the file

Since:

1.5

PrintStream

```
public PrintStream(File file,
    String csn)
    throws FileNotFoundException,
    UnsupportedEncodingException
```

Creates a new print stream, without automatic line flushing, with the specified file and charset.

This convenience constructor creates the necessary intermediate [OutputStreamWriter](#), which will encode characters using the provided charset.

Parameters:

`file` - The file to use as the destination of this print stream. If the file exists, then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.

`csn` - The name of a supported [charset](#)

Throws:

[FileNotFoundException](#) - If the given file object does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file

[SecurityException](#) - If a security manager is present and [checkWrite\(file.getPath\(\)\)](#) denies write access to the file

[UnsupportedEncodingException](#) - If the named charset is not supported

Since:

1.5

Method Detail

flush

```
public void flush()
```

Flush the stream. This is done by writing any buffered output bytes to the underlying output stream and then flushing that stream.

Specified by:

[flush](#) in interface [Flushable](#)

Overrides:

[flush](#) in class [FilterOutputStream](#)

See Also:

[OutputStream.flush\(\)](#)

close

```
public void close()
```

Close the stream. This is done by flushing the stream and then closing the underlying output stream.

Specified by:[close](#) in interface [Closeable](#)**Overrides:**[close](#) in class [FilterOutputStream](#)**See Also:**[OutputStream.close\(\)](#)

checkError

```
public boolean checkError()
```

Flush the stream and check its error state. The internal error state is set to `true` when the underlying output stream throws an `IOException` other than `InterruptedException`, and when the `setError` method is invoked. If an operation on the underlying output stream throws an `InterruptedException`, then the `PrintStream` converts the exception back into an interrupt by doing:

```
Thread.currentThread().interrupt();
```

or the equivalent.

Returns:

True if and only if this stream has encountered an `IOException` other than `InterruptedException`, or the `setError` method has been invoked

setError

```
protected void setError()
```

Set the error state of the stream to `true`.

Since:

JDK1.1

write

```
public void write(int b)
```

Write the specified byte to this stream. If the byte is a newline and automatic flushing is enabled then the `flush` method will be invoked.

Note that the byte is written as given; to write a character that will be translated according to the platform's default character encoding, use the `print(char)` or `println(char)` methods.

Overrides:

[write](#) in class [FilterOutputStream](#)

Parameters:

b - The byte to be written

See Also:

[print\(char\)](#), [println\(char\)](#)

write

```
public void write(byte[] buf,  
                  int off,  
                  int len)
```

Write `len` bytes from the specified byte array starting at offset `off` to this stream. If automatic flushing is enabled then the `flush` method will be invoked.

Note that the bytes will be written as given; to write characters that will be translated according to the platform's default character encoding, use the `print(char)` or `println(char)` methods.

Overrides:

[write](#) in class [FilterOutputStream](#)

Parameters:

buf - A byte array

off - Offset from which to start taking bytes

len - Number of bytes to write

See Also:

[FilterOutputStream.write\(int\)](#)

print

```
public void print(boolean b)
```

Print a boolean value. The string produced by [String.valueOf\(boolean\)](#) is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the [write\(int\)](#) method.

Parameters:

b - The boolean to be printed

print

```
public void print(char c)
```

Print a character. The character is translated into one or more bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the [write\(int\)](#) method.

Parameters:

c - The char to be printed

print

```
public void print(int i)
```

Print an integer. The string produced by [String.valueOf\(int\)](#) is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the [write\(int\)](#) method.

Parameters:

i - The int to be printed

See Also:

[Integer.toString\(int\)](#)

print

```
public void print(long l)
```

Print a long integer. The string produced by [String.valueOf\(long\)](#) is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the [write\(int\)](#) method.

Parameters:

l - The long to be printed

See Also:

[Long.toString\(long\)](#)

print

```
public void print(float f)
```

Print a floating-point number. The string produced by [String.valueOf\(float\)](#) is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the [write\(int\)](#) method.

Parameters:

f - The float to be printed

See Also:

[Float.toString\(float\)](#)

print

```
public void print(double d)
```

Print a double-precision floating-point number. The string produced by [String.valueOf\(double\)](#) is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the [write\(int\)](#) method.

Parameters:

d - The double to be printed

See Also:

[Double.toString\(double\)](#)

print

```
public void print(char[] s)
```

Print an array of characters. The characters are converted into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the [write\(int\)](#) method.

Parameters:

s - The array of chars to be printed

Throws:

[NullPointerException](#) - If s is null

print

```
public void print(String s)
```

Print a string. If the argument is null then the string "null" is printed. Otherwise, the string's characters are converted into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the [write\(int\)](#) method.

Parameters:

s - The String to be printed

print

```
public void print(Object obj)
```

Print an object. The string produced by the [String.valueOf\(Object\)](#) method is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the [write\(int\)](#) method.

Parameters:

obj - The Object to be printed

See Also:

[Object.toString\(\)](#)

println

`public void println()`

Terminate the current line by writing the line separator string. The line separator string is defined by the system property `line.separator`, and is not necessarily a single newline character (`'\n'`).

`println`

`public void println(boolean x)`

Print a boolean and then terminate the line. This method behaves as though it invokes [print\(boolean\)](#) and then [println\(\)](#).

Parameters:

x - The boolean to be printed

`println`

`public void println(char x)`

Print a character and then terminate the line. This method behaves as though it invokes [print\(char\)](#) and then [println\(\)](#).

Parameters:

x - The char to be printed.

`println`

`public void println(int x)`

Print an integer and then terminate the line. This method behaves as though it invokes [print\(int\)](#) and then [println\(\)](#).

Parameters:

x - The int to be printed.

println

```
public void println(long x)
```

Print a long and then terminate the line. This method behaves as though it invokes [print\(long\)](#) and then [println\(\)](#).

Parameters:

x - a The long to be printed.

println

```
public void println(float x)
```

Print a float and then terminate the line. This method behaves as though it invokes [print\(float\)](#) and then [println\(\)](#).

Parameters:

x - The float to be printed.

println

```
public void println(double x)
```

Print a double and then terminate the line. This method behaves as though it invokes [print\(double\)](#) and then [println\(\)](#).

Parameters:

x - The double to be printed.

println

```
public void println(char[] x)
```

Print an array of characters and then terminate the line. This method behaves as though it invokes [print\(char\[\]\)](#) and then [println\(\)](#).

Parameters:

x - an array of chars to print.

println

```
public void println(String x)
```

Print a String and then terminate the line. This method behaves as though it invokes [print\(\[String\]\(#\)\)](#) and then [println\(\)](#).

Parameters:

x - The String to be printed.

println

```
public void println(Object x)
```

Print an Object and then terminate the line. This method behaves as though it invokes [print\(\[Object\]\(#\)\)](#) and then [println\(\)](#).

Parameters:

x - The Object to be printed.

printf

```
public PrintStream printf(String format,  
                           Object... args)
```

A convenience method to write a formatted string to this output stream using the specified format string and arguments.

An invocation of this method of the form `out.printf(format, args)` behaves in exactly the same way as the invocation

```
out.format(format, args)
```

Parameters:

format - A format string as described in [Format string syntax](#)

args - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The number of arguments is variable and may be zero. The maximum number of arguments is limited by the maximum dimension of a Java array as defined by the [Java Virtual Machine Specification](#). The behaviour on a `null` argument depends on the [conversion](#).

Returns:

This output stream

Throws:

IllegalArgumentException - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the [Details](#) section of the formatter class specification.

NullPointerException - If the format is `null`

Since:

1.5

printf

```
public PrintStream printf(Locale l,
                        String format,
                        Object... args)
```

A convenience method to write a formatted string to this output stream using the specified format string and arguments.

An invocation of this method of the form `out.printf(l, format, args)` behaves in exactly the same way as the invocation

```
out.format(l, format, args)
```

Parameters:

l - The [locale](#) to apply during formatting. If `l` is `null` then no localization is applied.

format - A format string as described in [Format string syntax](#)

args - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The number of arguments is variable and may be zero. The maximum number of arguments is limited by the maximum dimension of a Java array as defined by the [Java Virtual Machine Specification](#). The behaviour on a `null` argument depends on the [conversion](#).

Returns:

This output stream

Throws:

[IllegalFormatException](#) - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the [Details](#) section of the formatter class specification.

[NullPointerException](#) - If the format is null

Since:

1.5

format

```
public PrintStream format(String format,  
                           Object... args)
```

Writes a formatted string to this output stream using the specified format string and arguments.

The locale always used is the one returned by [Locale.getDefault\(\)](#), regardless of any previous invocations of other formatting methods on this object.

Parameters:

`format` - A format string as described in [Format string syntax](#)

`args` - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The number of arguments is variable and may be zero. The maximum number of arguments is limited by the maximum dimension of a Java array as defined by the [Java Virtual Machine Specification](#). The behaviour on a null argument depends on the [conversion](#).

Returns:

This output stream

Throws:

[IllegalFormatException](#) - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the [Details](#) section of the formatter class specification.

[NullPointerException](#) - If the format is null

Since:

1.5

format

```
public PrintStream format(Locale l,
                        String format,
                        Object... args)
```

Writes a formatted string to this output stream using the specified format string and arguments.

Parameters:

`l` - The [locale](#) to apply during formatting. If `l` is `null` then no localization is applied.
`format` - A format string as described in [Format string syntax](#)
`args` - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The number of arguments is variable and may be zero. The maximum number of arguments is limited by the maximum dimension of a Java array as defined by the [Java Virtual Machine Specification](#). The behaviour on a `null` argument depends on the [conversion](#).

Returns:

This output stream

Throws:

[IllegalFormatException](#) - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the [Details](#) section of the formatter class specification.
[NullPointerException](#) - If the format is `null`

Since:

1.5

append

```
public PrintStream append(CharSequence csq)
```

Appends the specified character sequence to this output stream.

An invocation of this method of the form `out.append(csq)` behaves in exactly the same way as the invocation

```
out.print(csq.toString())
```

Depending on the specification of `toString` for the character sequence `csq`, the entire sequence may not be appended. For instance, invoking then `toString` method of a character buffer will return a subsequence whose content depends upon the buffer's position

and limit.

Specified by:

[append](#) in interface [Appendable](#)

Parameters:

csq - The character sequence to append. If csq is null, then the four characters "null" are appended to this output stream.

Returns:

This character stream

Since:

1.5

append

```
public PrintStream append(char c)
```

Appends the specified character to this output stream.

An invocation of this method of the form `out.append(c)` behaves in exactly the same way as the invocation

```
out.print(c)
```

Specified by:

[append](#) in interface [Appendable](#)

Parameters:

c - The 16-bit character to append

Returns:

This output stream

Since:

1.5

[Overview](#) [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.5.0*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: FIELD | [CONSTR](#) | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

OBERON-2 COMPILERS, JAVA MHC CORP PASCAL MODULA-2

[Home](#) | [Java](#) | [Pascal](#) | [Modula-2](#) | [Oberon-2](#) | [Orders](#) | [Mail](#)



Last update Feb 24, 2003

java.io

Interface DataInput

All Known Subinterfaces:

[InputStream](#), [OutputStream](#), [ObjectInput](#)

All Known Implementing Classes:

[DataInputStream](#), [InputStreamImpl](#), [OutputStreamImpl](#), [ObjectInputStream](#),
[RandomAccessFile](#)public interface **DataInput**

The **DataInput** interface provides for reading bytes from a binary stream and reconstructing from them data in any of the Java primitive types. There is also a facility for reconstructing a **String** from data in Java modified UTF-8 format.

It is generally true of all the reading routines in this interface that if end of file is reached before the desired number of bytes has been read, an **EOFException** (which is a kind of **IOException**) is thrown. If any byte cannot be read for any reason other than end of file, an **IOException** other than **EOFException** is thrown. In particular, an **IOException** may be thrown if the input stream has been closed.

Since:

JDK1.0

See Also:

[DataInputStream](#), [DataOutput](#)

Method Summary

boolean	readBoolean() Reads one input byte and returns true if that byte is nonzero, false if that byte is zero.

byte	<code>readByte()</code> Reads and returns one input byte.
char	<code>readChar()</code> Reads an input <code>char</code> and returns the <code>char</code> value.
double	<code>readDouble()</code> Reads eight input bytes and returns a <code>double</code> value.
float	<code>readFloat()</code> Reads four input bytes and returns a <code>float</code> value.
void	<code>readFully(byte[] b)</code> Reads some bytes from an input stream and stores them into the buffer array <code>b</code> .
void	<code>readFully(byte[] b, int off, int len)</code> Reads <code>len</code> bytes from an input stream.
int	<code>readInt()</code> Reads four input bytes and returns an <code>int</code> value.
<code>String</code>	<code>readLine()</code> Reads the next line of text from the input stream.
long	<code>readLong()</code> Reads eight input bytes and returns a <code>long</code> value.
short	<code>readShort()</code> Reads two input bytes and returns a <code>short</code> value.
int	<code>readUnsignedByte()</code> Reads one input byte, zero-extends it to type <code>int</code> , and returns the result, which is therefore in the range 0 through 255.
int	<code>readUnsignedShort()</code> Reads two input bytes and returns an <code>int</code> value in the range 0 through 65535.
<code>String</code>	<code>readUTF()</code> Reads in a string that has been encoded using a modified UTF-8 format.
int	<code>skipBytes(int n)</code> Makes an attempt to skip over <code>n</code> bytes of data from the input stream, discarding the skipped bytes.

Method Detail

readFully

```
public void readFully(byte[] b)
    throws IOException
```

Reads some bytes from an input stream and stores them into the buffer array `b`. The number of bytes read is equal to the length of `b`.

This method blocks until one of the following conditions occurs:

- `b.length` bytes of input data are available, in which case a normal return is made.
- End of file is detected, in which case an `EOFException` is thrown.
- An I/O error occurs, in which case an `IOException` other than `EOFException` is thrown.

If `b` is `null`, a `NullPointerException` is thrown. If `b.length` is zero, then no bytes are read. Otherwise, the first byte read is stored into element `b[0]`, the next one into `b[1]`, and so on. If an exception is thrown from this method, then it may be that some but not all bytes of `b` have been updated with data from the input stream.

Parameters:

`b` - the buffer into which the data is read.

Throws:

[EOFException](#) - if this stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

readFully

```
public void readFully(byte[] b,
    int off,
    int len)
    throws IOException
```

Reads `len` bytes from an input stream.

This method blocks until one of the following conditions occurs:

- `len` bytes of input data are available, in which case a normal return is made.
- End of file is detected, in which case an `EOFException` is thrown.
- An I/O error occurs, in which case an `IOException` other than `EOFException` is thrown.

If `b` is `null`, a `NullPointerException` is thrown. If `off` is negative, or `len` is negative, or `off+len` is greater than the length of the array `b`, then an

`IndexOutOfBoundsException` is thrown. If `len` is zero, then no bytes are read. Otherwise, the first byte read is stored into element `b[off]`, the next one into `b[off+1]`, and so on. The number of bytes read is, at most, equal to `len`.

Parameters:

`b` - the buffer into which the data is read.

`off` - an int specifying the offset into the data.

`len` - an int specifying the number of bytes to read.

Throws:

[EOFException](#) - if this stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

skipBytes

```
public int skipBytes(int n)  
    throws IOException
```

Makes an attempt to skip over `n` bytes of data from the input stream, discarding the skipped bytes. However, it may skip over some smaller number of bytes, possibly zero. This may result from any of a number of conditions; reaching end of file before `n` bytes have been skipped is only one possibility. This method never throws an `EOFException`. The actual number of bytes skipped is returned.

Parameters:

`n` - the number of bytes to be skipped.

Returns:

the number of bytes actually skipped.

Throws:

[IOException](#) - if an I/O error occurs.

readBoolean

```
public boolean readBoolean()  
    throws IOException
```

Reads one input byte and returns `true` if that byte is nonzero, `false` if that byte is zero. This method is suitable for reading the byte written by the `writeBoolean` method of interface `DataOutput`.

Returns:

the boolean value read.

Throws:

[EOFException](#) - if this stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

readByte

```
public byte readByte()  
           throws IOException
```

Reads and returns one input byte. The byte is treated as a signed value in the range -128 through 127, inclusive. This method is suitable for reading the byte written by the `writeByte` method of interface `DataOutput`.

Returns:

the 8-bit value read.

Throws:

[EOFException](#) - if this stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

readUnsignedByte

```
public int readUnsignedByte()  
           throws IOException
```

Reads one input byte, zero-extends it to type `int`, and returns the result, which is therefore in the range 0 through 255. This method is suitable for reading the byte written by the `writeByte` method of interface `DataOutput` if the argument to `writeByte` was intended to be a value in the range 0 through 255.

Returns:

the unsigned 8-bit value read.

Throws:

[EOFException](#) - if this stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

readShort

```
public short readShort()
           throws IOException
```

Reads two input bytes and returns a `short` value. Let `a` be the first byte read and `b` be the second byte. The value returned is:

$$(\text{short})((a \ll 8) * | (b \& 0xff))$$

This method is suitable for reading the bytes written by the `writeShort` method of interface `DataOutput`.

Returns:

the 16-bit value read.

Throws:

[EOFException](#) - if this stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

readUnsignedShort

```
public int readUnsignedShort()
           throws IOException
```

Reads two input bytes and returns an `int` value in the range 0 through 65535. Let `a` be the first byte read and `b` be the second byte. The value returned is:

$$(((a \& 0xff) \ll 8) | (b \& 0xff))$$

This method is suitable for reading the bytes written by the `writeShort` method of interface `DataOutput` if the argument to `writeShort` was intended to be a value in the range 0 through 65535.

Returns:

the unsigned 16-bit value read.

Throws:

[EOFException](#) - if this stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

readChar

```
public char readChar()
    throws IOException
```

Reads an input `char` and returns the `char` value. A Unicode `char` is made up of two bytes. Let `a` be the first byte read and `b` be the second byte. The value returned is:

$$(\text{char})((a \ll 8) \mid (b \& 0\text{xff}))$$

This method is suitable for reading bytes written by the `writeChar` method of interface `DataOutput`.

Returns:

the Unicode `char` read.

Throws:

[EOFException](#) - if this stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

readInt

```
public int readInt()
    throws IOException
```

Reads four input bytes and returns an `int` value. Let `a` be the first byte read, `b` be the second byte, `c` be the third byte, and `d` be the fourth byte. The value returned is:

$$(((a \& 0\text{xff}) \ll 24) \mid ((b \& 0\text{xff}) \ll 16) \mid ((c \& 0\text{xff}) \ll 8) \mid (d \& 0\text{xff}))$$

This method is suitable for reading bytes written by the `writeInt` method of interface `DataOutput`.

Returns:

the `int` value read.

Throws:

[EOFException](#) - if this stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

readLong

```
public long readLong()  
    throws IOException
```

Reads eight input bytes and returns a `long` value. Let `a` be the first byte read, `b` be the second byte, `c` be the third byte, `d` be the fourth byte, `e` be the fifth byte, `f` be the sixth byte, `g` be the seventh byte, and `h` be the eighth byte. The value returned is:

```
((long)(a & 0xff) << 56) |  
((long)(b & 0xff) << 48) |  
((long)(c & 0xff) << 40) |  
((long)(d & 0xff) << 32) |  
((long)(e & 0xff) << 24) |  
((long)(f & 0xff) << 16) |  
((long)(g & 0xff) << 8) |  
((long)(h & 0xff))
```

This method is suitable for reading bytes written by the `writeLong` method of interface `DataOutput`.

Returns:

the `long` value read.

Throws:

[EOFException](#) - if this stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

readFloat

```
public float readFloat()  
    throws IOException
```

Reads four input bytes and returns a `float` value. It does this by first constructing an `int` value in exactly the manner of the `readInt` method, then converting this `int` value to a `float` in exactly the manner of the method `Float.intBitsToFloat`. This method is suitable for reading bytes written by the `writeFloat` method of interface `DataOutput`.

Returns:

the `float` value read.

Throws:

[EOFException](#) - if this stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

readDouble

```
public double readDouble()  
    throws IOException
```

Reads eight input bytes and returns a `double` value. It does this by first constructing a `long` value in exactly the manner of the `readLong` method, then converting this `long` value to a `double` in exactly the manner of the method `Double.longBitsToDouble`. This method is suitable for reading bytes written by the `writeDouble` method of interface `DataOutput`.

Returns:

the `double` value read.

Throws:

[EOFException](#) - if this stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

readLine

```
public String readLine()  
    throws IOException
```

Reads the next line of text from the input stream. It reads successive bytes, converting each byte separately into a character, until it encounters a line terminator or end of file; the characters read are then returned as a `String`. Note that because this method processes bytes, it does not support input of the full Unicode character set.

If end of file is encountered before even one byte can be read, then `null` is returned. Otherwise, each byte that is read is converted to type `char` by zero-extension. If the character `'\n'` is encountered, it is discarded and reading ceases. If the character `'\r'` is encountered, it is discarded and, if the following byte converts to the character `'\n'`, then that is discarded also; reading then ceases. If end of file is encountered before either of the characters `'\n'` and `'\r'` is encountered, reading ceases. Once reading has ceased, a `String` is returned that contains all the characters read and not discarded, taken in order. Note that every character in this string will have a value less than `\u0100`, that is, `(char)256`.

Returns:

the next line of text from the input stream, or `null` if the end of file is encountered before a byte can be read.

Throws:

[IOException](#) - if an I/O error occurs.

readUTF

```
public String readUTF()  
    throws IOException
```

Reads in a string that has been encoded using a modified UTF-8 format. The general contract of `readUTF` is that it reads a representation of a Unicode character string encoded in Java modified UTF-8 format; this string of characters is then returned as a `String`.

First, two bytes are read and used to construct an unsigned 16-bit integer in exactly the manner of the `readUnsignedShort` method. This integer value is called the *UTF length* and specifies the number of additional bytes to be read. These bytes are then converted to characters by considering them in groups. The length of each group is computed from the value of the first byte of the group. The byte following a group, if any, is the first byte of the next group.

If the first byte of a group matches the bit pattern `0xxxxxxx` (where `x` means "may be 0 or 1"), then the group consists of just that byte. The byte is zero-extended to form a character.

If the first byte of a group matches the bit pattern `110xxxxx`, then the group consists of that byte `a` and a second byte `b`. If there is no byte `b` (because byte `a` was the last of the bytes to be read), or if byte `b` does not match the bit pattern `10xxxxxx`, then a `UTFDataFormatException` is thrown. Otherwise, the group is converted to the character:

$$(\text{char})(((a \& 0x1F) \ll 6) \mid (b \& 0x3F))$$

If the first byte of a group matches the bit pattern `1110xxxx`, then the group consists of that byte `a` and two more bytes `b` and `c`. If there is no byte `c` (because byte `a` was one of the last two of the bytes to be read), or either byte `b` or byte `c` does not match the bit pattern `10xxxxxx`, then a `UTFDataFormatException` is thrown. Otherwise, the group is converted to the character:

$$(\text{char})(((a \& 0x0F) \ll 12) \mid ((b \& 0x3F) \ll 6) \mid (c \& 0x3F))$$

If the first byte of a group matches the pattern `1111xxxx` or the pattern `10xxxxxx`, then a `UTFDataFormatException` is thrown.

If end of file is encountered at any time during this entire process, then an `EOFException` is thrown.

After every group has been converted to a character by this process, the characters are gathered, in the same order in which their corresponding groups were read from the input stream, to form a `String`, which is returned.

The `writeUTF` method of interface `DataOutput` may be used to write data that is suitable for reading by this method.

Returns:

a Unicode string.

Throws:

[EOFException](#) - if this stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

[UTFDataFormatException](#) - if the bytes do not represent a valid UTF-8 encoding of a string.

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.4.2*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

java.io

Interface DataOutput

All Known Subinterfaces:
[ImageOutputStream](#), [ObjectOutput](#)

All Known Implementing Classes:
[DataOutputStream](#), [ImageOutputStreamImpl](#), [ObjectOutputStream](#), [RandomAccessFile](#)

public interface **DataOutput**

The `DataOutput` interface provides for converting data from any of the Java primitive types to a series of bytes and writing these bytes to a binary stream. There is also a facility for converting a `String` into Java modified UTF-8 format and writing the resulting series of bytes.

For all the methods in this interface that write bytes, it is generally true that if a byte cannot be written for any reason, an `IOException` is thrown.

Since:
JDK1.0

See Also:
[DataInput](#), [DataOutputStream](#)

Method Summary	
void	write (byte[] b) Writes to the output stream all the bytes in array b.
void	write (byte[] b, int off, int len) Writes len bytes from array b, in order, to the output stream.
void	write (int b) Writes to the output stream the eight low-order bits of the argument b.

void	writeBoolean (boolean v) Writes a boolean value to this output stream.
void	writeByte (int v) Writes to the output stream the eight low- order bits of the argument v.
void	writeBytes (String s) Writes a string to the output stream.
void	writeChar (int v) Writes a char value, wich is comprised of two bytes, to the output stream.
void	writeChars (String s) Writes every character in the string S, to the output stream, in order, two bytes per character.
void	writeDouble (double v) Writes a double value, which is comprised of eight bytes, to the output stream.
void	writeFloat (float v) Writes a float value, which is comprised of four bytes, to the output stream.
void	writeInt (int v) Writes an int value, which is comprised of four bytes, to the output stream.
void	writeLong (long v) Writes a long value, which is comprised of eight bytes, to the output stream.
void	writeShort (int v) Writes two bytes to the output stream to represent the value of the argument.
void	writeUTF (String str) Writes two bytes of length information to the output stream, followed by the Java modified UTF representation of every character in the string S.

Method Detail

write

```
public void write(int b)
    throws IOException
```

Writes to the output stream the eight low-order bits of the argument **b**. The 24 high-order bits of **b** are ignored.

Parameters:

b - the byte to be written.

Throws:

[IOException](#) - if an I/O error occurs.

write

```
public void write(byte[] b)
    throws IOException
```

Writes to the output stream all the bytes in array `b`. If `b` is `null`, a `NullPointerException` is thrown. If `b.length` is zero, then no bytes are written. Otherwise, the byte `b[0]` is written first, then `b[1]`, and so on; the last byte written is `b[b.length-1]`.

Parameters:

`b` - the data.

Throws:

[IOException](#) - if an I/O error occurs.

write

```
public void write(byte[] b,
    int off,
    int len)
    throws IOException
```

Writes `len` bytes from array `b`, in order, to the output stream. If `b` is `null`, a `NullPointerException` is thrown. If `off` is negative, or `len` is negative, or `off+len` is greater than the length of the array `b`, then an `IndexOutOfBoundsException` is thrown. If `len` is zero, then no bytes are written. Otherwise, the byte `b[off]` is written first, then `b[off+1]`, and so on; the last byte written is `b[off+len-1]`.

Parameters:

`b` - the data.

`off` - the start offset in the data.

`len` - the number of bytes to write.

Throws:

[IOException](#) - if an I/O error occurs.

writeBoolean

```
public void writeBoolean(boolean v)
           throws IOException
```

Writes a `boolean` value to this output stream. If the argument `v` is `true`, the value (byte) 1 is written; if `v` is `false`, the value (byte) 0 is written. The byte written by this method may be read by the `readBoolean` method of interface `DataInput`, which will then return a `boolean` equal to `v`.

Parameters:

`v` - the boolean to be written.

Throws:

[IOException](#) - if an I/O error occurs.

writeByte

```
public void writeByte(int v)
           throws IOException
```

Writes to the output stream the eight low- order bits of the argument `v`. The 24 high-order bits of `v` are ignored. (This means that `writeByte` does exactly the same thing as `write` for an integer argument.) The byte written by this method may be read by the `readByte` method of interface `DataInput`, which will then return a byte equal to (byte)`v`.

Parameters:

`v` - the byte value to be written.

Throws:

[IOException](#) - if an I/O error occurs.

writeShort

```
public void writeShort(int v)
           throws IOException
```

Writes two bytes to the output stream to represent the value of the argument. The byte values to be written, in the order shown, are:

```
(byte)(0xff & (v >> 8))
```

```
(byte)(0xff & v)
```

The bytes written by this method may be read by the `readShort` method of interface `DataInput` , which will then return a `short` equal to `(short)v`.

Parameters:

`v` - the `short` value to be written.

Throws:

[IOException](#) - if an I/O error occurs.

writeChar

```
public void writeChar(int v)
           throws IOException
```

Writes a `char` value, which is comprised of two bytes, to the output stream. The byte values to be written, in the order shown, are:

```
(byte)(0xff & (v >> 8))
(byte)(0xff & v)
```

The bytes written by this method may be read by the `readChar` method of interface `DataInput` , which will then return a `char` equal to `(char)v`.

Parameters:

`v` - the `char` value to be written.

Throws:

[IOException](#) - if an I/O error occurs.

writeInt

```
public void writeInt(int v)
           throws IOException
```

Writes an `int` value, which is comprised of four bytes, to the output stream. The byte values to be written, in the order shown, are:

```
(byte)(0xff & (v >> 24))
```



```
(byte)(0xff & (v >> 16))
(byte)(0xff & (v >> 8))
(byte)(0xff & v)
```

The bytes written by this method may be read by the `readInt` method of interface `DataInput` , which will then return an `int` equal to `v`.

Parameters:

`v` - the `int` value to be written.

Throws:

[IOException](#) - if an I/O error occurs.

writeLong

```
public void writeLong(long v)
           throws IOException
```

Writes a `long` value, which is comprised of eight bytes, to the output stream. The byte values to be written, in the order shown, are:

```
(byte)(0xff & (v >> 56))
(byte)(0xff & (v >> 48))
(byte)(0xff & (v >> 40))
(byte)(0xff & (v >> 32))
(byte)(0xff & (v >> 24))
(byte)(0xff & (v >> 16))
(byte)(0xff & (v >> 8))
(byte)(0xff & v)
```

The bytes written by this method may be read by the `readLong` method of interface `DataInput` , which will then return a `long` equal to `v`.

Parameters:

`v` - the `long` value to be written.

Throws:

[IOException](#) - if an I/O error occurs.

writeFloat

```
public void writeFloat(float v)
    throws IOException
```

Writes a `float` value, which is comprised of four bytes, to the output stream. It does this as if it first converts this `float` value to an `int` in exactly the manner of the `Float.floatToIntBits` method and then writes the `int` value in exactly the manner of the `writeInt` method. The bytes written by this method may be read by the `readFloat` method of interface `DataInput`, which will then return a `float` equal to `v`.

Parameters:

`v` - the `float` value to be written.

Throws:

[IOException](#) - if an I/O error occurs.

writeDouble

```
public void writeDouble(double v)
    throws IOException
```

Writes a `double` value, which is comprised of eight bytes, to the output stream. It does this as if it first converts this `double` value to a `long` in exactly the manner of the `Double.doubleToLongBits` method and then writes the `long` value in exactly the manner of the `writeLong` method. The bytes written by this method may be read by the `readDouble` method of interface `DataInput`, which will then return a `double` equal to `v`.

Parameters:

`v` - the `double` value to be written.

Throws:

[IOException](#) - if an I/O error occurs.

writeBytes

```
public void writeBytes(String s)
    throws IOException
```

Writes a string to the output stream. For every character in the string `s`, taken in order, one byte is written to the output stream. If `s` is `null`, a `NullPointerException` is thrown.

If `s.length` is zero, then no bytes are written. Otherwise, the character `s[0]` is written

first, then `s[1]`, and so on; the last character written is `s[s.length-1]`. For each character, one byte is written, the low-order byte, in exactly the manner of the `writeByte` method. The high-order eight bits of each character in the string are ignored.

Parameters:

`s` - the string of bytes to be written.

Throws:

[IOException](#) - if an I/O error occurs.

writeChars

```
public void writeChars(String s)
                throws IOException
```

Writes every character in the string `s`, to the output stream, in order, two bytes per character. If `s` is `null`, a `NullPointerException` is thrown. If `s.length` is zero, then no characters are written. Otherwise, the character `s[0]` is written first, then `s[1]`, and so on; the last character written is `s[s.length-1]`. For each character, two bytes are actually written, high-order byte first, in exactly the manner of the `writeChar` method.

Parameters:

`s` - the string value to be written.

Throws:

[IOException](#) - if an I/O error occurs.

writeUTF

```
public void writeUTF(String str)
                throws IOException
```

Writes two bytes of length information to the output stream, followed by the Java modified UTF representation of every character in the string `s`. If `s` is `null`, a `NullPointerException` is thrown. Each character in the string `s` is converted to a group of one, two, or three bytes, depending on the value of the character.

If a character `c` is in the range `\u0001` through `\u007f`, it is represented by one byte:

```
(byte)c
```

If a character `c` is `\u0000` or is in the range `\u0080` through `\u07ff`, then it is represented by two bytes, to be written in the order shown:

```
(byte)(0xc0 | (0x1f & (c >> 6)))
(byte)(0x80 | (0x3f & c))
```

If a character `c` is in the range `\u0800` through `uffff`, then it is represented by three bytes, to be written in the order shown:

```
(byte)(0xe0 | (0x0f & (c >> 12)))
(byte)(0x80 | (0x3f & (c >> 6)))
(byte)(0x80 | (0x3f & c))
```

First, the total number of bytes needed to represent all the characters of `s` is calculated. If this number is larger than 65535, then a `UTFDataFormatException` is thrown. Otherwise, this length is written to the output stream in exactly the manner of the `writeShort` method; after this, the one-, two-, or three-byte representation of each character in the string `s` is written.

The bytes written by this method may be read by the `readUTF` method of interface `DataInput`, which will then return a `String` equal to `s`.

Parameters:

`str` - the string value to be written.

Throws:

[IOException](#) - if an I/O error occurs.

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.4.2*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

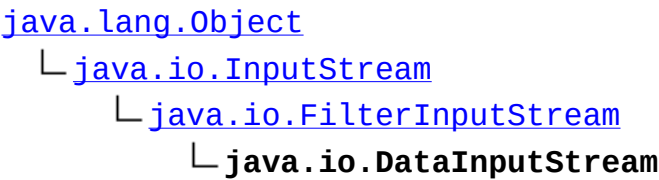
SUMMARY: NESTED | FIELD | CONSTR | [METHOD](#) DETAIL: FIELD | CONSTR | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

java.io
Class DataInputStream



All Implemented Interfaces:
[DataInput](#)

public class **DataInputStream**
extends [FilterInputStream](#)
implements [DataInput](#)

A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream.

Data input streams and data output streams represent Unicode strings in a format that is a slight modification of UTF-8. (For more information, see X/Open Company Ltd., "File System Safe UCS Transformation Format (FSS_UTF)", X/Open Preliminary Specification, Document Number: P316. This information also appears in ISO/IEC 10646, Annex P.) Note that in the following tables, the most significant bit appears in the far left-hand column.

All characters in the range '\u0001' to '\u007F' are represented by a single byte:

	Bit Values		
Byte 1	0	bits 6-0	

The null character '\u0000' and characters in the range '\u0080' to '\u07FF' are represented by a pair of bytes:

	Bit Values			
Byte 1	1	1	0	bits 10-6
Byte 2	1	0	bits 5-0	

Characters in the range '\u0800' to '\uFFFF' are represented by three bytes:

	Bit Values				
Byte 1	1	1	1	0	bits 15-12

Byte 2	1	0	bits 11-6
Byte 3	1	0	bits 5-0

The two differences between this format and the "standard" UTF-8 format are the following:

- The null byte '`\u0000`' is encoded in 2-byte format rather than 1-byte, so that the encoded strings never have embedded nulls.
- Only the 1-byte, 2-byte, and 3-byte formats are used.

Since:

JDK1.0

See Also:

[DataOutputStream](#)

Field Summary

Fields inherited from class [java.io.FilterInputStream](#)

[in](#)

Constructor Summary

[DataInputStream](#)([InputStream](#) in)

Creates a DataInputStream that uses the specified underlying InputStream.

Method Summary

int	read (byte[] b) Reads some number of bytes from the contained input stream and stores them into the buffer array b.

int	read (byte[] b, int off, int len) Reads up to len bytes of data from the contained input stream into an array of bytes.

boolean	readBoolean () See the general contract of the readBoolean method of DataInput.

byte	readByte () See the general contract of the readByte method of DataInput.

char	readChar () See the general contract of the readChar method of DataInput.

double	readDouble () See the general contract of the readDouble method of DataInput.

float	readFloat () See the general contract of the readFloat method of DataInput.

void	readFully (byte[] b) See the general contract of the readFully method of DataInput.

void	readFully (byte[] b, int off, int len) See the general contract of the <code>readFully</code> method of <code>DataInput</code> .
int	readInt () See the general contract of the <code>readInt</code> method of <code>DataInput</code> .
String	readLine () Deprecated. <i>This method does not properly convert bytes to characters. As of JDK 1.1, the preferred way to read lines of text is via the <code>BufferedReader.readLine()</code> method. Programs that use the <code>DataInputStream</code> class to read lines can be converted to use the <code>BufferedReader</code> class by replacing code of the form:</i> <i><code>DataInputStream d = new DataInputStream(in);</code></i> with: <i><code>BufferedReader d = new BufferedReader(new InputStreamReader(in));</code></i>
long	readLong () See the general contract of the <code>readLong</code> method of <code>DataInput</code> .
short	readShort () See the general contract of the <code>readShort</code> method of <code>DataInput</code> .
int	readUnsignedByte () See the general contract of the <code>readUnsignedByte</code> method of <code>DataInput</code> .
int	readUnsignedShort () See the general contract of the <code>readUnsignedShort</code> method of <code>DataInput</code> .
String	readUTF () See the general contract of the <code>readUTF</code> method of <code>DataInput</code> .
static String	readUTF (<code>DataInput</code> in) Reads from the stream <code>in</code> a representation of a Unicode character string encoded in Java modified UTF-8 format; this string of characters is then returned as a <code>String</code> .
int	skipBytes (int n) See the general contract of the <code>skipBytes</code> method of <code>DataInput</code> .

Methods inherited from class [java.io.FilterInputStream](#)[available](#), [close](#), [mark](#), [markSupported](#), [read](#), [reset](#), [skip](#)**Methods inherited from class [java.lang.Object](#)**[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)**Constructor Detail**

DataInputStream

```
public DataInputStream(InputStream in)
```

Creates a DataInputStream that uses the specified underlying InputStream.

Parameters:

`in` - the specified input stream

Method Detail

read

```
public final int read(byte[] b)
    throws IOException
```

Reads some number of bytes from the contained input stream and stores them into the buffer array `b`. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.

If `b` is null, a `NullPointerException` is thrown. If the length of `b` is zero, then no bytes are read and `0` is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value `-1` is returned; otherwise, at least one byte is read and stored into `b`.

The first byte read is stored into element `b[0]`, the next one into `b[1]`, and so on. The number of bytes read is, at most, equal to the length of `b`. Let `k` be the number of bytes actually read; these bytes will be stored in elements `b[0]` through `b[k-1]`, leaving elements `b[k]` through `b[b.length-1]` unaffected.

If the first byte cannot be read for any reason other than end of file, then an `IOException` is thrown. In particular, an `IOException` is thrown if the input stream has been closed.

The `read(b)` method has the same effect as:

```
read(b, 0, b.length)
```

Overrides:

[read](#) in class [FilterInputStream](#)

Parameters:

`b` - the buffer into which the data is read.

Returns:

the total number of bytes read into the buffer, or `-1` if there is no more data because the end of the stream has been reached.

Throws:

[IOException](#) - if an I/O error occurs.

See Also:

[FilterInputStream.in](#), [InputStream.read\(byte\[\], int, int\)](#)

read


```
public final int read(byte[] b,
                    int off,
                    int len)
    throws IOException
```

Reads up to `len` bytes of data from the contained input stream into an array of bytes. An attempt is made to read as many as `len` bytes, but a smaller number may be read, possibly zero. The number of bytes actually read is returned as an integer.

This method blocks until input data is available, end of file is detected, or an exception is thrown.

If `b` is `null`, a `NullPointerException` is thrown.

If `off` is negative, or `len` is negative, or `off+len` is greater than the length of the array `b`, then an `IndexOutOfBoundsException` is thrown.

If `len` is zero, then no bytes are read and `0` is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value `-1` is returned; otherwise, at least one byte is read and stored into `b`.

The first byte read is stored into element `b[off]`, the next one into `b[off+1]`, and so on. The number of bytes read is, at most, equal to `len`. Let `k` be the number of bytes actually read; these bytes will be stored in elements `b[off]` through `b[off+k-1]`, leaving elements `b[off+k]` through `b[off+len-1]` unaffected.

In every case, elements `b[0]` through `b[off]` and elements `b[off+len]` through `b[b.length-1]` are unaffected.

If the first byte cannot be read for any reason other than end of file, then an `IOException` is thrown. In particular, an `IOException` is thrown if the input stream has been closed.

Overrides:

[read](#) in class [FilterInputStream](#)

Parameters:

`b` - the buffer into which the data is read.
`off` - the start offset of the data.
`len` - the maximum number of bytes read.

Returns:

the total number of bytes read into the buffer, or `-1` if there is no more data because the end of the stream has been reached.

Throws:

[IOException](#) - if an I/O error occurs.

See Also:

[FilterInputStream.in](#), [InputStream.read\(byte\[\], int, int\)](#)

readFully

```
public final void readFully(byte[] b)
    throws IOException
```

See the general contract of the `readFully` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:

[readFully](#) in interface [DataInput](#)

Parameters:

b - the buffer into which the data is read.

Throws:

[EOFException](#) - if this input stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

See Also:

[FilterInputStream.in](#)

readFully

```
public final void readFully(byte[] b,  
                           int off,  
                           int len)  
    throws IOException
```

See the general contract of the `readFully` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:

[readFully](#) in interface [DataInput](#)

Parameters:

b - the buffer into which the data is read.

off - the start offset of the data.

len - the number of bytes to read.

Throws:

[EOFException](#) - if this input stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

See Also:

[FilterInputStream.in](#)

skipBytes

```
public final int skipBytes(int n)  
    throws IOException
```

See the general contract of the `skipBytes` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:

[skipBytes](#) in interface [DataInput](#)

Parameters:

n - the number of bytes to be skipped.

Returns:

the actual number of bytes skipped.

Throws:

[IOException](#) - if an I/O error occurs.

readBoolean

```
public final boolean readBoolean()  
                        throws IOException
```

See the general contract of the `readBoolean` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:

[readBoolean](#) in interface [DataInput](#)

Returns:

the boolean value read.

Throws:

[EOFException](#) - if this input stream has reached the end.

[IOException](#) - if an I/O error occurs.

See Also:

[FilterInputStream.in](#)

readByte

```
public final byte readByte()  
                  throws IOException
```

See the general contract of the `readByte` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:

[readByte](#) in interface [DataInput](#)

Returns:

the next byte of this input stream as a signed 8-bit byte.

Throws:

[EOFException](#) - if this input stream has reached the end.

[IOException](#) - if an I/O error occurs.

See Also:

[FilterInputStream.in](#)

readUnsignedByte

```
public final int readUnsignedByte()  
                throws IOException
```

See the general contract of the `readUnsignedByte` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:

[readUnsignedByte](#) in interface [DataInput](#)

Returns:

the next byte of this input stream, interpreted as an unsigned 8-bit number.

Throws:

[EOFException](#) - if this input stream has reached the end.

[IOException](#) - if an I/O error occurs.

See Also:

[FilterInputStream.in](#)

readShort

```
public final short readShort()  
                throws IOException
```

See the general contract of the `readShort` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:

[readShort](#) in interface [DataInput](#)

Returns:

the next two bytes of this input stream, interpreted as a signed 16-bit number.

Throws:

[EOFException](#) - if this input stream reaches the end before reading two bytes.

[IOException](#) - if an I/O error occurs.

See Also:

[FilterInputStream.in](#)

readUnsignedShort

```
public final int readUnsignedShort()  
                throws IOException
```

See the general contract of the `readUnsignedShort` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:

[readUnsignedShort](#) in interface [DataInput](#)

Returns:

the next two bytes of this input stream, interpreted as an unsigned 16-bit integer.

Throws:

[EOFException](#) - if this input stream reaches the end before reading two bytes.

[IOException](#) - if an I/O error occurs.

See Also:

[FilterInputStream.in](#)

readChar

```
public final char readChar()  
                throws IOException
```

See the general contract of the `readChar` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:

[readChar](#) in interface [DataInput](#)

Returns:

the next two bytes of this input stream as a Unicode character.

Throws:

[EOFException](#) - if this input stream reaches the end before reading two bytes.

[IOException](#) - if an I/O error occurs.

See Also:

[FilterInputStream.in](#)

readInt

```
public final int readInt()  
                throws IOException
```

See the general contract of the `readInt` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:

[readInt](#) in interface [DataInput](#)

Returns:

the next four bytes of this input stream, interpreted as an `int`.

Throws:

[EOFException](#) - if this input stream reaches the end before reading four bytes.

[IOException](#) - if an I/O error occurs.

See Also:

[FilterInputStream.in](#)

readLong

```
public final long readLong()  
    throws IOException
```

See the general contract of the `readLong` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:

[readLong](#) in interface [DataInput](#)

Returns:

the next eight bytes of this input stream, interpreted as a `long`.

Throws:

[EOFException](#) - if this input stream reaches the end before reading eight bytes.

[IOException](#) - if an I/O error occurs.

See Also:

[FilterInputStream.in](#)

readFloat

```
public final float readFloat()  
    throws IOException
```

See the general contract of the `readFloat` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:

[readFloat](#) in interface [DataInput](#)

Returns:

the next four bytes of this input stream, interpreted as a `float`.

Throws:

[EOFException](#) - if this input stream reaches the end before reading four bytes.

[IOException](#) - if an I/O error occurs.

See Also:

[readInt\(\)](#), [Float.intBitsToFloat\(int\)](#)

readDouble

```
public final double readDouble()  
    throws IOException
```

See the general contract of the `readDouble` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:[readDouble](#) in interface [DataInput](#)**Returns:**

the next eight bytes of this input stream, interpreted as a double.

Throws:[EOFException](#) - if this input stream reaches the end before reading eight bytes.[IOException](#) - if an I/O error occurs.**See Also:**[readLong\(\)](#), [Double.longBitsToDouble\(long\)](#)**readLine**

```
public final String readLine()
           throws IOException
```

Deprecated. *This method does not properly convert bytes to characters. As of JDK 1.1, the preferred way to read lines of text is via the `BufferedReader.readLine()` method. Programs that use the `DataInputStream` class to read lines can be converted to use the `BufferedReader` class by replacing code of the form:*

```
DataInputStream d = new DataInputStream(in);
```

with:

```
BufferedReader d
    = new BufferedReader(new InputStreamReader(in));
```

See the general contract of the `readLine` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:[readLine](#) in interface [DataInput](#)**Returns:**

the next line of text from this input stream.

Throws:[IOException](#) - if an I/O error occurs.**See Also:**[BufferedReader.readLine\(\)](#), [FilterInputStream.in](#)**readUTF**

```
public final String readUTF()
           throws IOException
```

See the general contract of the `readUTF` method of `DataInput`.

Bytes for this operation are read from the contained input stream.

Specified by:

[readUTF](#) in interface [DataInput](#)

Returns:

a Unicode string.

Throws:

[EOFException](#) - if this input stream reaches the end before reading all the bytes.

[IOException](#) - if an I/O error occurs.

[UTFDataFormatException](#) - if the bytes do not represent a valid UTF-8 encoding of a string.

See Also:

[readUTF\(java.io.DataInput\)](#)

readUTF

```
public static final String readUTF(DataInput in)
                                throws IOException
```

Reads from the stream `in` a representation of a Unicode character string encoded in Java modified UTF-8 format; this string of characters is then returned as a `String`. The details of the modified UTF-8 representation are exactly the same as for the `readUTF` method of `DataInput`.

Parameters:

`in` - a data input stream.

Returns:

a Unicode string.

Throws:

[EOFException](#) - if the input stream reaches the end before all the bytes.

[IOException](#) - if an I/O error occurs.

[UTFDataFormatException](#) - if the bytes do not represent a valid Java modified UTF-8 encoding of a Unicode string.

See Also:

[readUnsignedShort\(\)](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.4.2*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: FIELD | [CONSTR](#) | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

[Overview](#) [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)*Java™ 2 Platform
Std. Ed. v1.4.2*[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#) [All Classes](#)SUMMARY: NESTED | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

java.io

Class `DataOutputStream`

[java.lang.Object](#)└─ [java.io.OutputStream](#)└─ [java.io.FilterOutputStream](#)└─ [java.io.DataOutputStream](#)

All Implemented Interfaces:

[DataOutput](#)public class **`DataOutputStream`**extends [FilterOutputStream](#)implements [DataOutput](#)

A data output stream lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in.

Since:

JDK1.0

See Also:

[DataInputStream](#)

Field Summary

protected int	written The number of bytes written to the data output stream so far.

Fields inherited from class java.io.[FilterOutputStream](#)

[out](#)

Constructor Summary

[DataOutputStream](#)([OutputStream](#) out)

Creates a new data output stream to write data to the specified underlying output stream.

Method Summary

void	flush () Flushes this data output stream.
int	size () Returns the current value of the counter <code>written</code> , the number of bytes written to this data output stream so far.
void	write (byte[] b, int off, int len) Writes len bytes from the specified byte array starting at offset off to the underlying output stream.
void	write (int b) Writes the specified byte (the low eight bits of the argument b) to the underlying output stream.
void	writeBoolean (boolean v) Writes a boolean to the underlying output stream as a 1-byte value.
void	writeByte (int v) Writes out a byte to the underlying output stream as a 1-byte value.
void	writeBytes (String s) Writes out the string to the underlying output stream as a sequence of bytes.
void	writeChar (int v) Writes a char to the underlying output stream as a 2-byte value, high byte first.
void	writeChars (String s) Writes a string to the underlying output stream as a sequence of characters.
void	writeDouble (double v) Converts the double argument to a long using the <code>doubleToLongBits</code> method in class <code>Double</code> , and then writes that long value to the underlying output stream as an 8-byte quantity, high byte first.
void	writeFloat (float v) Converts the float argument to an int using the <code>floatToIntBits</code> method in class <code>Float</code> , and then writes that int value to the underlying output stream as a 4-byte quantity, high byte first.
void	writeInt (int v) Writes an int to the underlying output stream as four bytes, high byte first.

void	writeLong (long v) Writes a <code>long</code> to the underlying output stream as eight bytes, high byte first.
void	writeShort (int v) Writes a <code>short</code> to the underlying output stream as two bytes, high byte first.
void	writeUTF (String str) Writes a string to the underlying output stream using Java modified UTF-8 encoding in a machine-independent manner.

Methods inherited from class [java.io.FilterOutputStream](#)

[close](#), [write](#)

Methods inherited from class [java.lang.Object](#)

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

Methods inherited from interface [java.io.DataOutput](#)

[write](#)

Field Detail

written

protected int **written**

The number of bytes written to the data output stream so far. If this counter overflows, it will be wrapped to `Integer.MAX_VALUE`.

Constructor Detail

DataOutputStream

public **DataOutputStream**([OutputStream](#) out)

Creates a new data output stream to write data to the specified underlying output stream. The counter `written` is set to zero.

Parameters:

`out` - the underlying output stream, to be saved for later use.

See Also:

[FilterOutputStream.out](#)

Method Detail**write**

```
public void write(int b)
           throws IOException
```

Writes the specified byte (the low eight bits of the argument `b`) to the underlying output stream. If no exception is thrown, the counter `written` is incremented by 1.

Implements the `write` method of `OutputStream`.

Specified by:

[write](#) in interface [DataOutput](#)

Overrides:

[write](#) in class [FilterOutputStream](#)

Parameters:

`b` - the byte to be written.

Throws:

[IOException](#) - if an I/O error occurs.

See Also:

[FilterOutputStream.out](#)

write

```
public void write(byte[] b,
                  int off,
                  int len)
           throws IOException
```

Writes `len` bytes from the specified byte array starting at offset `off` to the underlying output stream. If no exception is thrown, the counter `written` is incremented by `len`.

Specified by:

[write](#) in interface [DataOutput](#)

Overrides:[write](#) in class [FilterOutputStream](#)**Parameters:**

b - the data.

off - the start offset in the data.

len - the number of bytes to write.

Throws:[IOException](#) - if an I/O error occurs.**See Also:**[FilterOutputStream.out](#)

flush

```
public void flush()
    throws IOException
```

Flushes this data output stream. This forces any buffered output bytes to be written out to the stream.

The `flush` method of `DataOutputStream` calls the `flush` method of its underlying output stream.

Overrides:[flush](#) in class [FilterOutputStream](#)**Throws:**[IOException](#) - if an I/O error occurs.**See Also:**[FilterOutputStream.out](#), [OutputStream.flush\(\)](#)

writeBoolean

```
public final void writeBoolean(boolean v)
    throws IOException
```

Writes a `boolean` to the underlying output stream as a 1-byte value. The value `true` is written out as the value (byte)1; the value `false` is written out as the value (byte)0. If no exception is thrown, the counter `written` is incremented by 1.

Specified by:

[writeBoolean](#) in interface [DataOutput](#)

Parameters:

`v` - a boolean value to be written.

Throws:

[IOException](#) - if an I/O error occurs.

See Also:

[FilterOutputStream.out](#)

writeByte

```
public final void writeByte(int v)
                        throws IOException
```

Writes out a `byte` to the underlying output stream as a 1-byte value. If no exception is thrown, the counter `written` is incremented by 1.

Specified by:

[writeByte](#) in interface [DataOutput](#)

Parameters:

`v` - a byte value to be written.

Throws:

[IOException](#) - if an I/O error occurs.

See Also:

[FilterOutputStream.out](#)

writeShort

```
public final void writeShort(int v)
                        throws IOException
```

Writes a `short` to the underlying output stream as two bytes, high byte first. If no exception is thrown, the counter `written` is incremented by 2.

Specified by:

[writeShort](#) in interface [DataOutput](#)

Parameters:

`v` - a short to be written.

Throws:

[IOException](#) - if an I/O error occurs.

See Also:[FilterOutputStream.out](#)

writeChar

```
public final void writeChar(int v)
                        throws IOException
```

Writes a `char` to the underlying output stream as a 2-byte value, high byte first. If no exception is thrown, the counter `written` is incremented by 2.

Specified by:[writeChar](#) in interface [DataOutput](#)**Parameters:**

`v` - a `char` value to be written.

Throws:[IOException](#) - if an I/O error occurs.**See Also:**[FilterOutputStream.out](#)

writeInt

```
public final void writeInt(int v)
                        throws IOException
```

Writes an `int` to the underlying output stream as four bytes, high byte first. If no exception is thrown, the counter `written` is incremented by 4.

Specified by:[writeInt](#) in interface [DataOutput](#)**Parameters:**

`v` - an `int` to be written.

Throws:[IOException](#) - if an I/O error occurs.**See Also:**[FilterOutputStream.out](#)

writeLong

```
public final void writeLong(long v)  
    throws IOException
```

Writes a `long` to the underlying output stream as eight bytes, high byte first. In no exception is thrown, the counter `written` is incremented by 8.

Specified by:

[writeLong](#) in interface [DataOutput](#)

Parameters:

`v` - a `long` to be written.

Throws:

[IOException](#) - if an I/O error occurs.

See Also:

[FilterOutputStream.out](#)

writeFloat

```
public final void writeFloat(float v)  
    throws IOException
```

Converts the `float` argument to an `int` using the `floatToIntBits` method in class `Float`, and then writes that `int` value to the underlying output stream as a 4-byte quantity, high byte first. If no exception is thrown, the counter `written` is incremented by 4.

Specified by:

[writeFloat](#) in interface [DataOutput](#)

Parameters:

`v` - a `float` value to be written.

Throws:

[IOException](#) - if an I/O error occurs.

See Also:

[FilterOutputStream.out](#), [Float.floatToIntBits\(float\)](#)

writeDouble

```
public final void writeDouble(double v)  
    throws IOException
```


Converts the double argument to a long using the `doubleToLongBits` method in class `Double`, and then writes that long value to the underlying output stream as an 8-byte quantity, high byte first. If no exception is thrown, the counter `written` is incremented by 8.

Specified by:

[writeDouble](#) in interface [DataOutput](#)

Parameters:

`v` - a double value to be written.

Throws:

[IOException](#) - if an I/O error occurs.

See Also:

[FilterOutputStream.out](#), [Double.doubleToLongBits\(double\)](#)

writeBytes

```
public final void writeBytes(String s)
                        throws IOException
```

Writes out the string to the underlying output stream as a sequence of bytes. Each character in the string is written out, in sequence, by discarding its high eight bits. If no exception is thrown, the counter `written` is incremented by the length of `s`.

Specified by:

[writeBytes](#) in interface [DataOutput](#)

Parameters:

`s` - a string of bytes to be written.

Throws:

[IOException](#) - if an I/O error occurs.

See Also:

[FilterOutputStream.out](#)

writeChars

```
public final void writeChars(String s)
                        throws IOException
```

Writes a string to the underlying output stream as a sequence of characters. Each character is written to the data output stream as if by the `writeChar` method. If no exception is thrown, the counter `written` is incremented by twice the length of `s`.

Specified by:

[writeChars](#) in interface [DataOutput](#)

Parameters:

`s` - a `String` value to be written.

Throws:

[IOException](#) - if an I/O error occurs.

See Also:

[writeChar\(int\)](#), [FilterOutputStream.out](#)

writeUTF

```
public final void writeUTF(String str)
                        throws IOException
```

Writes a string to the underlying output stream using Java modified UTF-8 encoding in a machine-independent manner.

First, two bytes are written to the output stream as if by the `writeShort` method giving the number of bytes to follow. This value is the number of bytes actually written out, not the length of the string. Following the length, each character of the string is output, in sequence, using the modified UTF-8 encoding for the character. If no exception is thrown, the counter `written` is incremented by the total number of bytes written to the output stream. This will be at least two plus the length of `str`, and at most two plus thrice the length of `str`.

Specified by:

[writeUTF](#) in interface [DataOutput](#)

Parameters:

`str` - a string to be written.

Throws:

[IOException](#) - if an I/O error occurs.

size

```
public final int size()
```

Returns the current value of the counter `written`, the number of bytes written to this data output stream so far. If the counter overflows, it will be wrapped to `Integer.MAX_VALUE`.

Returns:

the value of the `written` field.

See Also:

[`written`](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.4.2*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).


[my sf.net](#)
[software map](#)
[donate to sf.net](#)
[about sf.net](#)

Go

[Login via SSL](#)
[New User via SSL](#)

Search

SF.net Subscription

- [Subscribe Now](#)
- [Manage Subscription](#)
- [Advanced Search](#)
- [Direct Download](#)
- [Priority Tech Support](#)
- [Project Monitoring](#)

SF.net Resources

- [Site Docs](#)
- [Site Status](#) (06/08)
- [Site Map](#)
- [SF.net Supporters](#)
- [Compile Farm](#)
- [Foundries](#)
- [Project Help Wanted](#)
- [New Releases](#)
- [Get Support](#)

Site Sponsors



Most Active

- 1 [Gaim](#)
- 2 [Azureus - BitTorrent Client](#)
- 3 [eGroupWare: Enterprise Collaboration](#)
- 4 [ABC \[Yet Another Bittorrent Client\]](#)
- 5 [phpMyAdmin](#)
- 6 [WinMerge](#)
- 7 [XOOPS Dynamic Web CMS](#)
- 8 [FCKeditor](#)
- 9 [PearPC - PowerPC Emulator](#)
- 10 [TightVNC](#)

[More Activity >>](#)

Project: Java Class File Editor: Summary

[Summary](#) |
 [Admin](#) |
 [Home Page](#) |
 [Forums](#) |
 [Tracker](#) |
 [Bugs](#) |
 [Support](#) |
 [Patches](#) |
 [RFE](#) |
 [Lists](#) |
 [Tasks](#) |
 [Docs](#) |
 [News](#) |
 [CVS](#) |
 [Files](#) |

Open a Java class file binary to view or edit strings, attributes, methods and generate readable reports similar to the javap utility. In built verifier checks changes before saving the file. Easy to use Java Swing GUI.


[Java Foundry](#)

- Development Status: [5 - Production/Stable](#)
- Intended Audience: [Developers](#), [Education](#)
- License: [GNU Library or Lesser General Public License \(LGPL\)](#)
- Natural Language: [English](#)
- Operating System: [OS Independent](#)
- Programming Language: [Java](#)
- Topic: [Software Development](#)

Project UNIX name: classeditor

Registered: 2003-09-04 22:45

Activity Percentile (last week): 81.1139%

View project activity [statistics](#)

View list of [RSS feeds](#) available for this project

Need support? See the [support instructions provided by this project](#).

Developer Info

Project Admins:
[tanmaykm](#)

Developers: 3
[\[View Members\]](#)

Latest File Releases

Package	Version	Date	Notes / Monitor	Download
classeditor	2.23	March 20, 2004	-	Download

[\[View ALL Project Files\]](#)

Public Areas

[Project Home Page](#)

Tracker

- [Bugs](#) (**0 open** / **0 total**)

Bug Tracking System

- [Support Requests](#) (**0 open** / **2 total**)

Tech Support Tracking System

- [Patches](#) (**0 open** / **0 total**)

Patch Tracking System

- [Feature Requests](#) (**1 open** / **1 total**)

Feature Request Tracking System

[Public Forums](#) (**4** messages in **2** forums)

[DocManager: Project Documentation](#)

[Mailing Lists](#) (**2 total**)

[Task Manager](#)

Latest News

Java ClassEditor 2.23 released

tanmaykm - 2004-03-21 06:58

[\[Read More/Comment\]](#)

Version 2.22 Released

tanmaykm - 2003-09-14 23:27

[\[Read More/Comment\]](#)

[\[News archive\]](#)

[\[Submit News\]](#)

Top Downloads

- [User Manuals](#)
- [XMLization](#)

- 1 [eMule](#)
- 2 [BitTorrent](#)
- 3 [Azureus - BitTorrent Client](#)
- 4 [DC++](#)
- 5 [ABC \[Yet Another Bittorrent Client\]](#)
- 6 [VirtualDub](#)
- 7 [CDex](#)
- 8 [guliverkli](#)
- 9 [Gaim](#)
- 10 [VisualBoyAdvance](#)

[CVS Repository](#) (**0** commits, **0** adds)

- [Browse CVS](#)

[More Statistics](#) > >

SF.net Services

- [Tech Jobs](#)
- [Online Books](#)
- [Comparison Shop](#)
- [Partner Product Offers](#)
- [Get Broadband](#)
- [IT Research Library](#)

Sponsored Content

Powered by [SourceForge\(tm\)](#) collaborative software development tools from [VA Software](#)

© Copyright 2004 - [OSDN](#) Open Source Development Network, All Rights Reserved

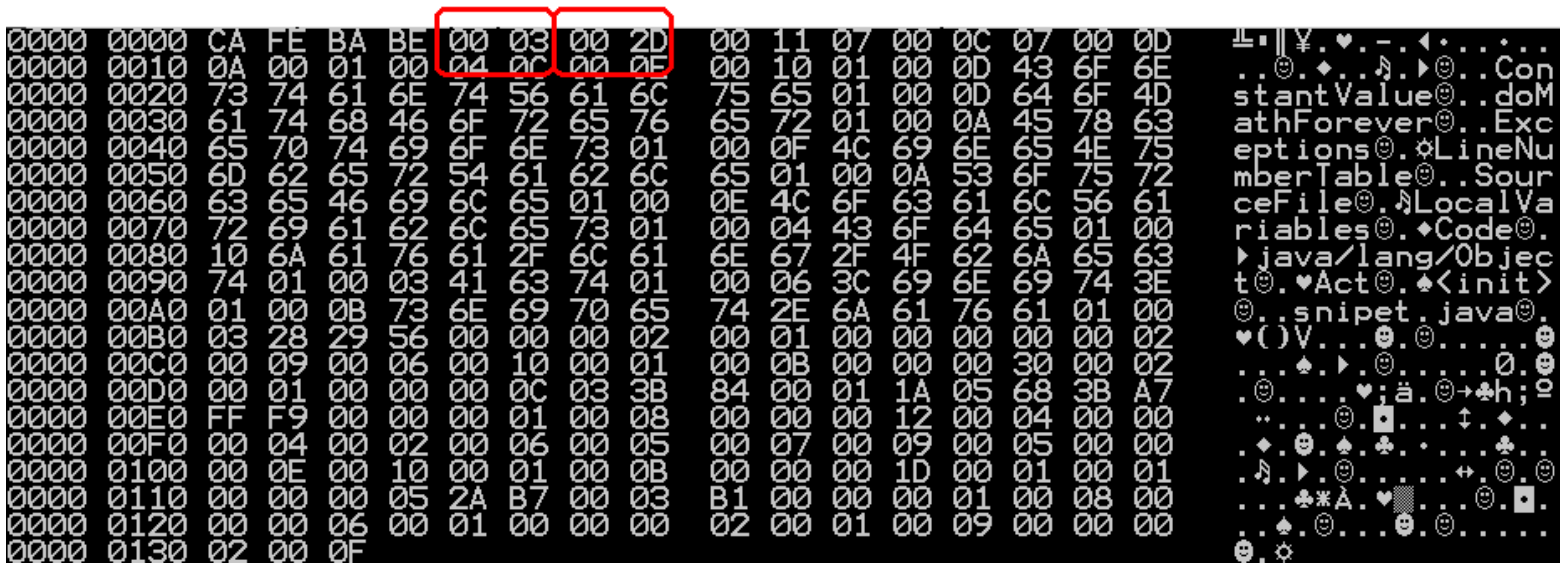
[About SourceForge.net](#) • [About OSDN](#) • [Privacy Statement](#) • [Terms of Use](#) • [Advertise](#) • [Self Serve Ad System](#) • [Get Support](#)

magic



0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	.. V.♥.-.◀...Con
0000	0010	8A	88	81	88	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	..@.♦...♪.▶@..doM
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	stantValue@..doM
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	athForever@..Exc
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	eptions@.*LineNu
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	mberTable@..Sour
0000	0060	63	65	46	69	6C	65	01	00	0E	4C	6F	63	61	6C	56	61	ceFile@.LocalVa
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	riables@.*Code@.
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	▶java/lang/Objec
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	t@.♥Act@.♠<init>
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	@..snipet.java@.
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	♥()V...@.@...@.
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	00	30	00	..♠.▶@.@...0.@.
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	@.@...♥:ä.@→♣h;@
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	@.@...@.♠...↑.♦..
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	.♦.@.♠.♣...♣..
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	.♪.▶@.@...↔.@.@.
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	01	00	08	00	@.@.*A.♥.♠...@.♠.
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	@.♠.@...@.@...@.
0000	0130	02	00	0F														@.♠.@...@.@...@.

version identifier
minor major





<http://www.jeckle.de/images/java/cf3.gif>08.06.2004 21:47:42



<http://www.jeckle.de/images/java/cf7.gif>08.06.2004 21:47:43

<http://www.jeckle.de/images/java/cf8.gif>08.06.2004 21:47:43

<http://www.jeckle.de/images/java/cf10.gif>08.06.2004 21:47:44

<http://www.jeckle.de/images/java/cf11.gif>08.06.2004 21:47:44

9. Element des constant_pool (Zeichenkette "SourceFile")

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	☪	☪	¥	.	♥	.	-	.	◀	.	.	.	.	.	.	.
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	..	☺	.	♦	.	♫	.	▶	☺	.	Con					
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	stantValue	☺	.	.	.	.	.	.	.	do	M					
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	athForever	☺	.	.	.	.	.	.	.	Exc						
0000	0040	65	70	74	69	6F	6F	73	01	00	0F	4C	69	6F	65	4F	75	eptions	☺	.	☼	LineNu											
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	mberTable	☺	.	☼	LocalVa											
0000	0060	63	65	46	69	6C	65	01	00	0E	4C	6F	63	61	6C	56	61	ceFile	☺	.	☼	LocalVa											
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	riables	☺	.	♦	Code											
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	▶java/lang/Objec															
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	t	☺	.	♥	Act	☺	.	♣	<init>							
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	☺	.	.	snipet.java	☺	.										
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	♥()	V	.	.	☺	.	☺	.	.	.	.	.	.	.	.	
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	30	00	02	.	.	♠	.	▶	☺	.	☺	.	0	.	☺	.			
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	.	☺	.	.	♥	.	ä	.	☺	→	♣	h	.	☺	.	
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	☺	.	☺	.	☺	.	☺	.	☺	.	☺	.	☺	.	☺	.
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	.	.	♦	.	☺	.	♠	.	☺	.	☺	.	☺	.	☺	.
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	♫	.	▶	☺	.	☺	.	☺	.	☺	.	☺	.	☺	.	
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	01	00	08	00	.	.	♣	*	Δ	.	♥	☼	.	.	☺	.	☺	.	☺	.
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	☺	.	☺	.	☺	.	☺	.	☺	.	☺	.	☺	.	☺	.
0000	0130	02	00	0F														☺	.	☺	.	☺	.	☺	.	☺	.	☺	.	☺	.	☺	.

10. Element des constant_pool (Zeichenkette "LocalVariable")

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	☪	☪	¥	♥	.	-	.	◀	.	.	.	.	.			
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	...	☺	.	♦	.	♪	.	▶	☺	.	Con					
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	stantValue	☺	.	do	M											
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	athForever	☺	.	Exc												
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	eptions	☺	.	☼	LineNu											
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	numberTable	☺	.	.	Sour											
0000	0060	63	65	46	69	6C	65	01	00	0F	4C	6F	63	61	6C	56	61	ceFile	☺	.	☼	LocalVa											
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	riables	☺	.	♦	Code	☺	.									
0000	0080	10	6A	61	78	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	▶java/lang/Objec															
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	t	☺	.	♥	Act	☺	.	♠	<init>							
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	☺	.	.	snippet.java	☺	.										
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	♥()	V	.	.	☺	.	☺	.	.	.	.					
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	30	00	02	.	.	♠	.	▶	☺	.	☺	.	0	☺					
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	.	☺	.	.	♥	.	ä	.	☺	→	♣	h	;	☺		
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	.	.	☺	.	☺	.	☺	.	☺	.	☺	.	☺	.	☺	
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	.	.	♦	.	☺	.	♠	.	☺	.	☺	.	☺	.	☺	
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	.	.	♣	.	▶	☺	.	☺	.	☺	.	☺	.	☺	.	☺
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	01	00	08	00	.	.	♣	.	*	.	♥	.	☺	.	☺	.	☺	.	☺	
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	.	.	☺	.	☺	.	☺	.	☺	.	☺	.	☺	.	☺	
0000	0130	02	00	0F														☺	.	☺	.	☺	.	☺	.	☺	.	☺	.	☺	.	☺	

11. Element des constant_pool (Zeichenkette "Code")

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	☪	☪	¥	.	♥	.	-	.	◀	.	.	.	.	.	.
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	...	☺	.	♦	.	.	♪	.	▶	☺	.	.	.	Con	
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	...	stantValue	☺	.	.	.	.	.	.	.	.	.	.	doM	
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	...	athForever	☺	.	.	.	.	.	.	.	.	.	.	Exc	
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	...	eptions	☺	.	.	.	.	.	.	.	.	.	.	LineNu	
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	...	numberTable	☺	.	.	.	.	.	.	.	.	.	.	Sour	
0000	0060	63	65	46	69	6C	65	01	00	0F	4C	6F	63	61	6C	56	61	...	ceFile	☺	.	.	.	.	.	.	.	.	.	.	LocalVa	
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	...	riables	☺	.	.	.	.	.	.	.	.	.	.	Code	
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	...	▶java/lang/Objec	☺	.	.	.	.	.	.	.	.	.	.	t	
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	...	♥Act	☺	.	.	.	.	.	.	.	.	.	.	<init>	
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	...	☺	.	.	.	.	.	.	.	.	.	.	.	snippet.java	
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	...	♥()	V	.	.	.	.	.	.	.	.	.	.	☺	
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	30	00	02	...	.	.	.	.	.	.	.	.	.	.	.	.	☺	
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	...	.	.	.	.	.	.	.	.	.	.	.	.	☺	
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	...	.	.	.	.	.	.	.	.	.	.	.	.	☺	
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	...	.	.	.	.	.	.	.	.	.	.	.	.	☺	
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	...	.	.	.	.	.	.	.	.	.	.	.	.	☺	
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	00	01	00	08	00	...	.	.	.	.	.	.	.	.	.	.	.	.	☺
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	...	.	.	.	.	.	.	.	.	.	.	.	.	☺	
0000	0130	02	00	0F														...	☺	.	.	.	.	.	.	.	.	.	.	.	☺	

<http://www.jeckle.de/images/java/cf16.gif>08.06.2004 21:47:45

<http://www.jeckle.de/images/java/cf17.gif>08.06.2004 21:47:46

<http://www.jeckle.de/images/java/cf19.gif>08.06.2004 21:47:46

Zugriffsflags

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	⌚	¥	♥	.	-	.	◀	.	.	.	.
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	..	☺	.	♦	.	♪	.	▶	☺	..	Con
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	stant	Value	☺	..	do	M					
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	ath	Forever	☺	..	Exc						
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	eptions	☺	.	✱	Line	Nu					
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	mber	Table	☺	..	Sour						
0000	0060	63	65	46	69	6C	65	01	00	0E	4C	6F	63	61	6C	56	61	ce	File	☺	.	♪	Local	Va				
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	riables	☺	.	♦	Code	☺	.				
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	▶	java/lang/Objec									
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	t	☺	.	♥	Act	☺	.	♠	<init>		
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	☺	..	snipet	.	java	☺	.				
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	♥()	V	...	☺	.	☺	.	.	.	.	
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	30	00	02	..	.	♠	.	▶	☺	.	.	.	.	
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	.	☺	.	.	.	.	.	.	.	.	
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	..	☺	.	.	.	.	.	.	.	.	
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	..	☺	.	.	.	.	.	.	.	.	
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	..	♠	.	.	.	.	.	.	.	.	
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	01	00	08	00	..	♣	.	.	.	.	.	.	.	.	
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	..	♣	.	.	.	.	.	.	.	.	
0000	0130	02	00	0F														☺	.	♠	.	.	.	.	.	.	.	

<http://www.jeckle.de/images/java/cf20a.gif>08.06.2004 21:47:46

interface_count und fields_count

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	..	¥.	♥.	-.	◀.	...	...
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	..	☺.	♦.	♪.	▶.	☺.	Con
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	stantValue	☺.	doM				
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	athForever	☺.	Exc				
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	eptions	☺.	☼.	LineNu			
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	mberTable	☺.	...	Sour			
0000	0060	63	65	46	69	6C	65	01	00	0E	4C	6F	63	61	6C	56	61	ceFile	☺.	♪.	LocalVa			
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	riables	☺.	♦.	Code			
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	▶java/lang/Objec						
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	t	☺.	♥.	Act	☺.	♠.	<init>
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	☺.	...	snippet.java	☺.			
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	02		♥()	V.	...	☺.	☺.	...	☺.
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	30	00	02	...	♠.	▶.	☺.	...	0.	☺.
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	☺.	...	♥.	ä.	☺.	→.	♣.
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	...	☺.	...	☺.	...	↑.	♦.
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	...	♦.	☺.	♠.	...	☺.	...
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	♪.	▶.	☺.	...	+	☺.	☺.
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	01	00	08	00	...	♣.	*	A.	♥.	☺.	☺.
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	☺.	♠.	☺.	...	☺.	☺.	...
0000	0130	02	00	0F														☺.	☺.	☺.	...	☺.	☺.	...

methods_count

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	..	¥.	♥.	-.	◀.	...	...
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	..	☺.	♦.	♪.	▶.	☺.	Con
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	stantValue	☺.	do	M			
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	athForever	☺.	Exc				
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	eptions	☺.	✱	LineNu			
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	mberTable	☺.	..	Sour			
0000	0060	63	65	46	69	6C	65	01	00	0E	4C	6F	63	61	6C	56	61	ceFile	☺.	♪	LocalVa			
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	riables	☺.	♦	Code	☺.		
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	▶java/lang/Objec						
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	t	☺.	♥	Act	☺.	♣	<init>
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	☺.	..	snippet.java	☺.			
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	♥()	V.	...	☺.	☺.	...	☺.
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	00	30	00	...	♣.	▶.	☺.	...	0.	☺.
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	..	☺.	...	♥.	ä.	☺.	→♣h;
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	☺.	...	☺.	...	☺.	...	☺.
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	♦.	☺.	♣.	...	☺.	...	☺.
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	♪.	▶.	☺.	...	☺.	...	☺.
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	01	00	08	00	...	♣.	*	Δ.	♥.	☺.	...
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	☺.	...	☺.	...	☺.	...	☺.
0000	0130	02	00	0F														☺.	...	☺.	...	☺.	...	☺.

methods_info (doMathForever())

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	..	¥.	♥.	-.	◀.	...	...
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	..	☺.	♦.	♪.	▶.	☺.	Con
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	stantValue	☺.	doM				
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	athForever	☺.	Exc				
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	eptions	☺.	☼.	LineNu			
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	mberTable	☺.	...	Sour			
0000	0060	63	65	46	69	6C	65	01	00	0E	4C	6F	63	61	6C	56	61	ceFile	☺.	☼.	LocalVa			
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	riables	☺.	♦.	Code			
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	▶java/lang/Objec						
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	t	☺.	♥.	Act	☺.	♣.	<init>
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	☺.	...	snippet.java	☺.			
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	♥()	V.	...	☺.	☺.	...	☺.
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	30	00	02	...	♣.	▶.	☺.	...	0.	☺.
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	..	☺.	...	♥.	ä.	☺.	→♣h;
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	☺.	...	☺.	...	☺.	...	☺.
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	♦.	☺.	♣.	...	☺.	...	☺.
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	♪.	▶.	☺.	...	☺.	...	☺.
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	01	00	08	00	...	♣.	*	Δ.	♥.	☺.	...
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	☺.	...	☺.	...	☺.	...	☺.
0000	0130	02	00	0F														☺.	...	☺.	...	☺.	...	☺.

<http://www.jeckle.de/images/java/cf24.gif>08.06.2004 21:47:48

maxStack und maxLocals der Methode doMathForever()

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	..	¥.	♥.	-.	◀.	...	...	
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	..	☺.	♦.	...	♪.	▶.	☺.	Con
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	stantValue	☺.	doM					
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	athForever	☺.	Exc					
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	eptions	☺.	☼.	LineNu				
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	mberTable	☺.	...	Sour				
0000	0060	63	65	46	69	6C	65	01	00	0E	4C	6F	63	61	6C	56	61	ceFile	☺.	...	LocalVa				
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	riables	☺.	...	Code				
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	▶java/lang/Objec							
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	t	☺.	♥.	Act	☺.	...	◀.	init>
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	☺.	...	snippet.java	☺.				
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	♥()	V.	...	☺.	☺.	...	☺.	
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	30	00	02	...	...	...	...	...	...	...	
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	...	...	...	...	...	...	...	
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	☺.	...	☺.	...	...	...	...	
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	...	...	...	...	...	...	...	
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	...	...	...	...	...	...	...	
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	01	00	08	00	...	...	...	...	...	...	...	
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	...	...	...	...	...	...	...	
0000	0130	02	00	0F														☺.	...	...	...	...	...	...	

Bytecode und Ausnahmentabelle der Methode doMathForever()

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	..	¥.	♥.	-.	◀.	...	...
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	...	...	...	...	...	...	Con
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	stantValue@..doM						
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	athForever@..Exc						
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	eptions@.✱LineNu						
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	mberTable@..Sour						
0000	0060	63	65	46	69	6C	65	01	00	0E	4C	6F	63	61	6C	56	61	ceFile@.♠LocalVa						
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	riables@.♦Code@.						
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	▶java/lang/Objec						
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	t@.♥Act@.♠<init>						
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	@..snippet.java@.						
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	♥()V...@.@....						
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	30	00	02	...	♠.	▶.	...	...	...	0.
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	...	...	...	...	...	...	;
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	...	...	...	...	...	...	;
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	...	...	...	...	...	...	;
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	...	...	...	...	...	...	;
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	01	00	08	00	...	...	...	...	...	...	;
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	...	...	...	...	...	...	;
0000	0130	02	00	0F														...	...	...	...	...	...	;

Eigenschaften des Code-Bereichs der Methode doMathForever()

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	..	¥.	♥.	-.	◀.	...	...
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	...	...	...	...	...	...	Con
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	...	...	...	...	...	...	stantValue@..doM
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	...	...	...	...	...	...	athForever@..Exc
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	...	...	...	...	...	...	eptions@..*LineNu
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	...	...	...	...	...	...	mberTable@..Sour
0000	0060	63	65	46	69	6C	65	01	00	0E	4C	6F	63	61	6C	56	61	...	...	...	...	...	...	ceFile@..*LocalVa
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	...	...	...	...	...	...	riables@..*Code@.
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	...	...	...	...	...	...	▶java/lang/Objec
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	...	...	...	...	...	...	t@..♥Act@..*◀init>
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	...	...	...	...	...	...	@..snippet.java@.
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	...	...	...	...	...	...	♥()V...@..@...@
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	30	00	02	...	...	...	...	...	...	...@...@...@...@
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	...	...	...	...	...	...	...@...@...@...@
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	...	...	...	...	...	...	...@...@...@...@
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	...	...	...	...	...	...	...@...@...@...@
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	...	...	...	...	...	...	...@...@...@...@
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	01	00	08	00	...	...	...	...	...	...	...@...@...@...@
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	...	...	...	...	...	...	...@...@...@...@
0000	0130	02	00	0F														...	...	...	...	...	...	...@...@...@...@

Zugriffsflaggen und Indizes der Klasse

Act()

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	..	¥.	♥.	-.	◀.	...	...
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	...	...	...	...	...	...	Con
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	...	...	...	...	...	...	stantValue@..doM
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	...	...	...	...	...	...	athForever@..Exc
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	...	...	...	...	...	...	eptions@.✱LineNu
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	...	...	...	...	...	...	mberTable@..Sour
0000	0060	63	65	46	69	6C	65	01	00	0E	4C	6F	63	61	6C	56	61	...	...	...	...	...	...	ceFile@.♠LocalVa
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	...	...	...	...	...	...	riables@.♦Code@.
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	...	...	...	...	...	...	▶java/lang/Objec
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	...	...	...	...	...	...	t@.♥Act@.♠<init>
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	...	...	...	...	...	...	@..snippet.java@.
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	...	...	...	...	...	...	♥()V...@.@.@@.
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	30	00	02	...	...	...	...	...	...	...♠.▶.@.@.@@.
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	...	...	...	...	...	...	...@.@.@.ä.@.→♣h;@.
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	...	...	...	...	...	...	...@.@.@.@.@.@.@.@.
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	...	...	...	...	...	...	...♦.@.♠.♣.@.@.@.@.
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	...	...	...	...	...	...	...♠.▶.@.@.@.@.@.
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	01	00	08	00	...	...	...	...	...	...	...♣.♠.A.♥.♠.@.@.@.
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	...	...	...	...	...	...	...@.@.@.@.@.@.@.
0000	0130	02	00	0F														...	...	...	...	...	...	...@.@.@.@.@.@.@.

maxStack und maxLocals der Klasse

Act()

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	..	¥.	♥.	-.	◀.	...	...
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	...	...	...	...	...	...	Con
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	stantValue	...	...	...	...	...	doM
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	athForever	...	...	...	...	...	Exc
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	eptions	...	...	...	...	...	LineNu
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	mberTable	...	...	...	...	...	Sour
0000	0060	63	65	46	69	6C	65	01	00	0E	4C	6F	63	61	6C	56	61	ceFile	...	...	...	...	...	LocalVa
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	riables	...	...	...	...	...	Code
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	▶java/lang/Objec	...	...	...	...	...	t
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	t	...	...	...	...	...	Act
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	...	...	...	...	...	...	<init>
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	...	...	...	...	...	...	snipet.java
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	30	00	02	♥()	V...	...	...	...	...	...
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	...	...	...	...	...	...	...
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	...	...	...	...	...	...	...
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	...	...	...	...	...	...	...
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	...	...	...	...	...	...	...
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	01	00	08	00	...	...	...	...	...	...	...
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	...	...	...	...	...	...	...
0000	0130	02	00	0F														...	...	...	...	...	...	...

Bytecode und Ausnahmetabelle der Klasse

Act()

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	..	¥.	♥.	-.	◀.	...	...
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	...	...	...	...	...	...	Con
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	stantValue	...	doM	...	...	...	
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	athForever	...	Exc	...	...	...	
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	eptions	...	...	...	...	...	
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	numberTable	...	...	...	...	...	
0000	0060	63	65	46	69	6C	65	01	00	0E	4C	6F	63	61	6C	56	61	ceFile	...	...	...	...	...	
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	riables	...	...	...	...	...	
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	▶java/lang/Objec	...	...	...	...	...	
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	t	...	...	...	...	...	
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	...	...	...	...	...	...	
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	...	...	...	...	...	...	
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	30	00	02	...	...	...	...	...	...	
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	...	...	...	...	...	...	
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	...	...	...	...	...	...	
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	...	...	...	...	...	...	
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	...	...	...	...	...	...	
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	00	01	00	08	...	...	...	...	...	...	
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	...	...	...	...	...	...	
0000	0130	02	00	0F														...	...	...	...	...	...	

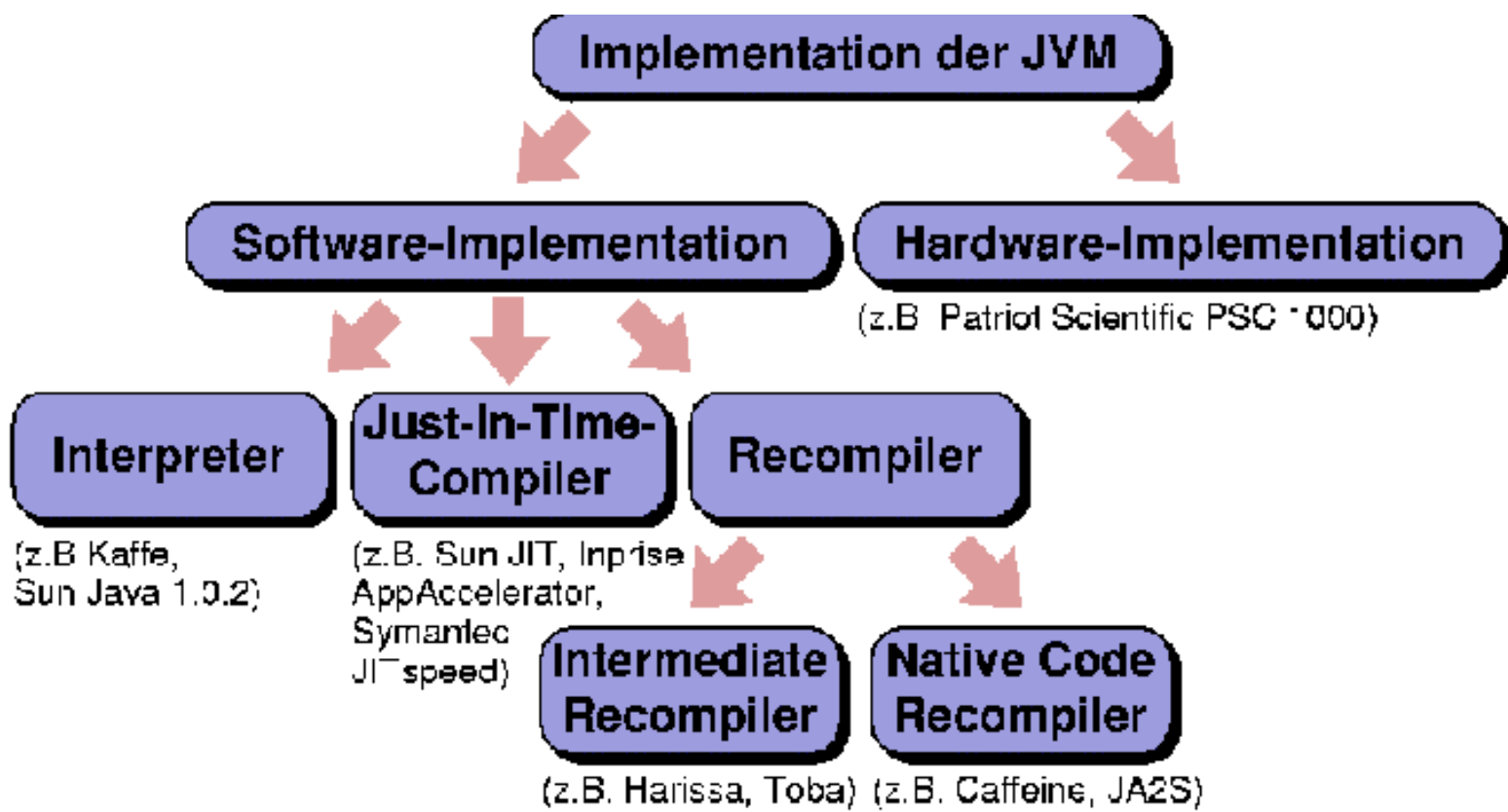
Eigenschaften des Code-Bereichs der Klasse Act()

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	..	¥.	♥.	-.	◀.	...	...
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	...	...	...	...	...	...	Con
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	stantValue	...	...	...	...	...	doM
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	athForever	...	...	...	...	...	Exc
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	eptions	...	...	...	...	...	LineNu
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	mberTable	...	...	...	...	...	Sour
0000	0060	63	65	46	69	6C	65	01	00	0E	4C	6F	63	61	6C	56	61	ceFile	...	...	...	...	...	LocalVa
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	riables	...	...	...	...	...	Code
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	▶java/lang/Objec	...	...	...	...	...	t
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	t	...	...	...	...	...	Act
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	...	...	...	...	...	...	<init>
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	...	...	...	...	...	...	snipet.java
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	30	00	02	♥()	V...	...	...	...	...	...
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	...	...	...	...	...	...	...
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	...	...	...	...	...	...	...
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	...	...	...	...	...	...	...
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	10	00	01	00	01	...	...	...	...	...	...	...
0000	0110	00	00	00	05	2A	B7	00	03	B1	00	00	00	01	00	08	00	...	...	...	...	...	...	...
0000	0120	00	00	06	00	01	00	00	00	02	00	01	00	09	00	00	00	...	...	...	...	...	...	...
0000	0130	02	00	0F														...	...	...	...	...	...	...

Zuordnung zwischen LineNumberTable und der Klasse Act()

0000	0000	CA	FE	BA	BE	00	03	00	2D	00	11	07	00	0C	07	00	0D	..	¥.	♥.	-.	◀.	...	...
0000	0010	0A	00	01	00	04	0C	00	0E	00	10	01	00	0D	43	6F	6E	...	...	...	...	...	...	Con
0000	0020	73	74	61	6E	74	56	61	6C	75	65	01	00	0D	64	6F	4D	stantValue	...	...	...	...	...	doM
0000	0030	61	74	68	46	6F	72	65	76	65	72	01	00	0A	45	78	63	athForever	...	...	...	...	...	Exc
0000	0040	65	70	74	69	6F	6E	73	01	00	0F	4C	69	6E	65	4E	75	eptions	...	...	...	...	...	LineNu
0000	0050	6D	62	65	72	54	61	62	6C	65	01	00	0A	53	6F	75	72	mberTable	...	...	...	...	...	Sour
0000	0060	63	65	46	69	6C	65	01	00	0E	4C	6F	63	61	6C	56	61	ceFile	...	...	...	...	...	LocalVa
0000	0070	72	69	61	62	6C	65	73	01	00	04	43	6F	64	65	01	00	riables	...	...	...	...	...	Code
0000	0080	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	▶java/lang/Objec	...	...	...	...	...	t
0000	0090	74	01	00	03	41	63	74	01	00	06	3C	69	6E	69	74	3E	t	...	...	...	...	...	Act
0000	00A0	01	00	0B	73	6E	69	70	65	74	2E	6A	61	76	61	01	00	...	...	...	...	...	...	<init>
0000	00B0	03	28	29	56	00	00	00	02	00	01	00	00	00	00	00	02	...	...	...	...	...	...	snipet.java
0000	00C0	00	09	00	06	00	10	00	01	00	0B	00	00	00	30	00	02	♥()	V...	...	...	...	...	...
0000	00D0	00	01	00	00	00	0C	03	3B	84	00	01	1A	05	68	3B	A7	...	...	...	...	...	...	...
0000	00E0	FF	F9	00	00	00	01	00	08	00	00	00	12	00	04	00	00	...	...	...	...	...	...	...
0000	00F0	00	04	00	02	00	06	00	05	00	07	00	09	00	05	00	00	...	...	...	...	...	...	...
0000	0100	00	0E	00	10	00	01	00	0B	00	00	00	1D	00	01	00	01	...	...	...	...	...	...	...
0000	0110	00	00	00	05	2A	87	00	03	81	00	00	00	01	00	08	00	...	...	...	...	...	...	...
0000	0120	00	00	06	00	01	00	00	02	00	01	00	09	00	00	00	00	...	...	...	...	...	...	...
0000	0130	02	00	0F														...	...	...	...	...	...	...

<http://www.jeckle.de/images/java/cf35.gif>08.06.2004 21:47:51



Inside the Java Virtual Machine JVM Resources

A list of recommended books covering topics covered by Inside the Java Virtual Machine is available at [Books Related to the JVM](#).

Java Class File Obfuscation

- [The Obfuscation Home Page](#) has many links to obfuscation and decompilation tools and information.
- [HashJava](#) is a free Java class file obfuscator.
- Neil Aggarwal's [Obfuscate and ObfuscatePro](#).
- PreEmptive Solutions, Inc.'s [DashO](#) Java class file optimizer and obfuscator.

Further Reading about Java Security

- [Secure Computing with Java: Now and the Future](#) (a whitepaper)
- [Applet Security FAQ](#)
- [Low Level Security in Java](#), by Frank Yellin
- The [Hostile Applets Home Page](#)
- This page has information about [authentication](#) in JDK 1.1.
- [Project Kimera](#) describes an interesting procedure for testing bytecode verifiers for correctness.

Further Reading about Network Mobility

Here are a few articles I wrote for JavaWorld that describe other forms of network-mobile Java code:

- [Java beyond the browser: The channel metaphor](#). This article explains what Marimba Castanet is and how it addresses some of the constraints presented by Java applets. It includes an example Castanet channel, Artima WordSurfer, that is a simple English vocabulary flashcards drill.
- [The Architecture of Aglets](#). This article explains the inner workings of aglets, IBM's Java-based autonomous software agent.
- [The Point of Aglets](#). This article examines the significance of the mobile

agent approach to developing network services. (This article will appear April 15th.)

An Audio Impression

Here's an applet that provides an audio impression of the way software feels now that it is free to zip across the network as class files. Just press the Flying Class Files button:

For some reason your browser won't let you view (or hear) this way cool Java applet.

Further Reading on the Java Virtual Machine

- For more information about the Java Virtual Machine, the first place to look is the specification. The book, *Java Virtual Machine Specification*, by Tim Lindholm and Frank Yellin is the official word. You can [browse](#) or [download](#) a free HTML version of this book at JavaSoft's web site.
- After the *Java Virtual Machine Specification*, you may want to refer to the book, *Java Language Specification*, by James Gosling, Bill Joy and Guy Steele. This book, which specifies the Java language, can give more insight into the inner workings of the Java Virtual Machine. You can [browse](#) or [download](#) a free HTML version of this book at JavaSoft's web site.

Newsgroups

- The newsgroup [comp.lang.java.machine](#) is dedicated to issues related to the Java Virtual Machine.

Further Reading on Class Files

- [Chapter 4, The Java Class File Format](#), from the *Java Virtual Machine Specification* by Tim Lindholm and Frank Yellin is also available online.
- The January, 1998 issue of *Doctor Dobb's Journal* contains an article on the Java class file written by Matt T. Yourst. This article, titled "[Inside the Java Class File](#)," is also published online .
- [Why CAFEBABE?](#) - an amusing discussion thread from comp . lang . java.

Tools for Viewing Class Files

- The JDK includes a tool, named [javap](#), which disassembles class files.

- [JavaDump](#) makes an HTML listing of the contents of a class file.

Tools for Manipulating Class Files

- [JavaStrip](#) strips out debug information from class files.

Libraries for Parsing/Generating Class Files

- [JAS](#), a freely Java bytecode assembler, includes source code for a library that generates Java class files.
- [JCF](#) is a freely available library for parsing Java class files
- Chuck McManis wrote and made available some [Java classes](#) that will dump out the contents of a Java class file.

Further Reading on the Lifetime of a Class

- The lifetime of a class is described in Section 2.16, [Execution](#), from Chapter 5, Constant Pool Resolution, of the *Java Virtual Machine Specification* by Tim Lindholm and Frank Yellin. .

Further Reading on the Linking Model

- [Chapter 5, Constant Pool Resolution](#), from the *Java Virtual Machine Specification* by Tim Lindholm and Frank Yellin is also available online. .
- Chuck McManis wrote an article for *JavaWorld*, named "[The Basics of Java Class Loaders](#)," that describes class loaders.

Further Reading about Garbage Collection

For an in-depth treatment of garbage collection, check out the book [Garbage Collection: Algorithms for Automatic Dynamic Memory Management](#) by Richard Jones and Rafael Lins.

Here are three papers that give good general overviews of garbage collection:

- Andrew W. Appel, "Garbage collection," in P. Lee (ed.), *Advanced Language Implementations*, MIT Press, 1991, chapter 4, pp. 89-100.
- Jacques Cohen, 'Garbage collection of linked data structures', *Computing Surveys*, 13(3), 1981, pp. 341-367.
- Paul R. Wilson, "Uniprocessor garbage collection techniques," Yves

Bekkers and Jacques Cohen (eds.), *Memory Management - International Workshop IWMM 92*, St. Malo, France, September 1992, pp. 1-42.

Further Reading on Bytecodes

- Chapter 10, [Opcode Mnemonics by Opcode](#), from the the *Java Virtual Machine Specification* lists the instruction set opcodes in numerical order.
- The [bytecode reference](#) in the O'Reilly JVM book is also available online.

Bytecode Assemblers

- [JAS](#) is a freely Java bytecode assembler that includes source code.
- [Jasmin](#) is a freely available Java bytecode assembler that is based on the JAS library.
- [Java Bytecode Assembler](#) whose input is similar to the output of javap. Use of this assembler is free for educational purposes only.

Further Reading on Threads and Monitors

- The form of monitor used by Java is just one of many different kinds of mechanisms that fall under the name "monitor". For an in-depth technical overview of monitors in general, check out this paper:

BUHR, Peter A., FORTIER, Michel, and COFFIN, Michael H. 1995. Monitor Classification. *ACM Computing Surveys*, Vol. 27, No. 1, (March 1995), 63-107

- For a good overview of all the mechanisms used in coordinating or synchronizing threads, including monitors, see:

ANDREWS, Gregory R., and SCHNEIDER, Fred B. 1983. Concepts and Notations for Concurrent Programming. *Computing Surveys*, Vol. 15, No. 1, (March 1983), 3-43

- For a complete description of the low-level rules Java Threads must follow, see [Chapter 8, Threads and Locks](#), from *The Java Virtual Machine Specification* by Tim Lindholm and Frank Yellin. This chapter is basically a reprint of [Chapter 17, Threads and Locks](#), from *The Java Language Specification* by James Gosling, Bill Joy, and Guy Steel. These chapters specify how threads must behave in every Java Virtual Machine implementation. The presentation is thick and mathematical. They are not

what I'd call easy reading, but as they are the specification, they give the full story. You can read them online by following the links above.

A proposal for revising the specification of Java Threads is at: <http://www.cs.umd.edu/~pugh/java/memoryModel/jsr.html>

Pavilion Pages

[Cluster with Coherence](#) for reliability, scalability and performance.

Visit the [Crystal Developer Zone](#) for Crystal Reports info.

JIRA: [world class issue tracking](#) as used by NASA, BP, 3M & more!

[Try RetroVue](#), the Total Recall journaling debugger from VisiComp.

[OrindaBuild](#) writes matching Java to run Oracle PL/SQL procedures.

Sponsored Links

Shape the future of Java[tm] technology. [Join the JCP](#) today.

Combine [Job Scheduling + JMS Messaging](#) with SwiftMQ.

Last Updated: Wednesday, May 26, 2004
[Copyright](#) © 1996-2004 Artima Software, Inc.
All Rights Reserved.

URL: <http://www.artima.com/insidejvm/resources/index.html>
Trouble with this page? Contact: webmaster@artima.com

NAME

fork - create a new process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION

The *fork()* function creates a new process. The new process (child process) is an exact copy of the calling process (parent process) except as detailed below.

- The child process has a unique process ID.
- The child process ID also does not match any active process group ID.
- The child process has a different parent process ID (that is, the process ID of the parent process).
- The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors refers to the same open file description with the corresponding file descriptor of the parent.
- The child process has its own copy of the parent's open directory streams. Each open directory stream in the child process may share directory stream positioning with the corresponding directory stream of the parent.
- The child process may have its own copy of the parent's message catalogue descriptors.
- The child process' values of *tms_utime*, *tms_stime*, *tms_cutime* and *tms_cstime* are set to 0.
- The time left until an alarm clock signal is reset to 0.

- All *semadj* values are cleared.
- File locks set by the parent process are not inherited by the child process.
- The set of signals pending for the child process is initialised to the empty set.
- Interval timers are reset in the child process.
- If the Semaphores option is supported, any semaphores that are open in the parent process will also be open in the child process.
- If the Process Memory Locking option is supported, the child process does not inherit any address space memory locks established by the parent process via calls to [*mlockall\(\)*](#) or [*mlock\(\)*](#).
- Memory mappings created in the parent are retained in the child process. MAP_PRIVATE mappings inherited from the parent will also be MAP_PRIVATE mappings in the child, and any modifications to the data in these mappings made by the parent prior to calling *fork()* will be visible to the child. Any modifications to the data in MAP_PRIVATE mappings made by the parent after *fork()* returns will be visible only to the parent. Modifications to the data in MAP_PRIVATE mappings made by the child will be visible only to the child.
- If the Process Scheduling option is supported, for the SCHED_FIFO and SCHED_RR scheduling policies, the child process inherits the policy and priority settings of the parent process during a *fork()* function. For other scheduling policies, the policy and priority settings on *fork()* are implementation-dependent.
- If the Timers option is supported, per-process timers created by the parent are not inherited by the child process.
- If the Message Passing option is supported, the child process has its own copy of the message queue descriptors of the parent. Each of the message descriptors of the child refers to the same open message queue description as the corresponding message descriptor of the parent.
- If the Asynchronous Input and Output option is supported, no asynchronous input or asynchronous output operations are inherited by the child process.

The inheritance of process characteristics not defined by this document is implementation-dependent. After *fork()*, both the parent and the child processes are capable of executing independently before either one terminates.

A process is created with a single thread. If a multi-threaded process calls *fork()*, the new process contains a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. Consequently, to avoid errors, the child process may only execute async-signal safe operations until such time as one of the *exec* functions is called. Fork handlers may be established by means of the [*pthread_atfork\(\)*](#) function in order to maintain application invariants across *fork()* calls.

RETURN VALUE

Upon successful completion, *fork()* returns 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, -1 is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

ERRORS

The *fork()* function will fail if:

[EAGAIN]

The system lacked the necessary resources to create another process, or the system-imposed limit on the total number of processes under execution system-wide or by a single user {CHILD_MAX} would be exceeded.

The *fork()* function may fail if:

[ENOMEM]

Insufficient storage space is available.

EXAMPLES

None.

APPLICATION USAGE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[*alarm\(\)*](#), [*exec*](#), [*fcntl\(\)*](#), [*semop\(\)*](#), [*signal\(\)*](#), [*times\(\)*](#), [*<sys/types.h>*](#), [*<unistd.h>*](#).

DERIVATION

Derived from Issue 1 of the SVID.

UNIX ® is a registered Trademark of The Open Group.

Copyright © 1997 The Open Group

[[Main Index](#) | [XSH](#) | [XCU](#) | [XBD](#) | [XCURSES](#) | [XNS](#)]

```
int main (int argc, char** argv) {
    printf("parent\n");
    int i, pid;

    if ((pid=fork()) == -1) {
        printf("error creating process\n");
        exit(1);
    }

    if (pid == 0) {
        for (i=0; i<10; i++) {
            sleep(1);
            printf("child %d\n",i);
        }
    }
    if (pid > 0) {
        printf("child's pid=%d\n",pid);
        for (i=0; i<10; i++) {
            sleep(1);
            printf("parent again %d\n",i);
        }
    }
    return 0;
}
```


NAME

wait, waitpid - wait for a child process to stop or terminate

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

DESCRIPTION

The *wait()* and [waitpid\(\)](#) functions allow the calling process to obtain status information pertaining to one of its child processes. Various options permit status information to be obtained for child processes that have terminated or stopped. If status information is available for two or more child processes, the order in which their status is reported is unspecified.

The *wait()* function will suspend execution of the calling thread until status information for one of its terminated child processes is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If more than one thread is suspended in *wait()* or [waitpid\(\)](#) awaiting termination of the same process, exactly one thread will return the process status at the time of the target process termination. If status information is available prior to the call to *wait()*, return will be immediate.

The [waitpid\(\)](#) function will behave identically to *wait()*, if the *pid* argument is (**pid_t**)-1 and the *options* argument is 0. Otherwise, its behaviour will be modified by the values of the *pid* and *options* arguments.

The *pid* argument specifies a set of child processes for which status is requested. The [waitpid\(\)](#) function will only return the status of a child process from this set:

- If *pid* is equal to (**pid_t**)-1, status is requested for any child process. In this respect, [waitpid\(\)](#) is then equivalent to *wait()*.

- If *pid* is greater than 0, it specifies the process ID of a single child process for which status is requested.
- If *pid* is 0, status is requested for any child process whose process group ID is equal to that of the calling process.
- If *pid* is less than (**pid_t**)-1, status is requested for any child process whose process group ID is equal to the absolute value of *pid*.

The *options* argument is constructed from the bitwise-inclusive OR of zero or more of the following flags, defined in the header [<sys/wait.h>](#<sys/wait.h>).

WCONTINUED

The [waitpid\(\)](#waitpid()) function will report the status of any continued child process specified by *pid* whose status has not been reported since it continued from a job control stop.

WNOHANG

The [waitpid\(\)](#waitpid()) function will not suspend execution of the calling thread if status is not immediately available for one of the child processes specified by *pid*.

WUNTRACED

The status of any child processes specified by *pid* that are stopped, and whose status has not yet been reported since they stopped, will also be reported to the requesting process.

If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN, and the process has no unwaited for children that were transformed into zombie processes, the calling thread will block until all of the children of the process containing the calling thread terminate, and *wait()* and [waitpid\(\)](#waitpid()) will fail and set *errno* to [ECHILD].

If *wait()* or [waitpid\(\)](#waitpid()) return because the status of a child process is available, these functions will return a value equal to the process ID of the child process. In this case, if the value of the argument *stat_loc* is not a null pointer, information will be stored in the location pointed to by *stat_loc*. If and only if the status returned is from a terminated child process that returned 0 from *main()* or passed 0 as the *status* argument to [_exit\(\)](#_exit()) or [exit\(\)](#exit()), the value stored at the location pointed to by *stat_loc* will be 0. Regardless of its value, this information may be interpreted using the following macros, which are defined in [<sys/wait.h>](#<sys/wait.h>) and evaluate to integral expressions; the *stat_val* argument is the integer value pointed to by *stat_loc*.

WIFEXITED(*stat_val*)

Evaluates to a non-zero value if status was returned for a child process that terminated normally.

WEXITSTATUS(*stat_val*)

If the value of WIFEXITED(*stat_val*) is non-zero, this macro evaluates to the

low-order 8 bits of the *status* argument that the child process passed to [_exit\(\)](#) or [exit\(\)](#), or the value the child process returned from *main()*.

WIFSIGNALED(*stat_val*)

Evaluates to non-zero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught (see [<signal.h>](#)).

WTERMSIG(*stat_val*)

If the value of WIFSIGNALED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED(*stat_val*)

Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

WSTOPSIG(*stat_val*)

If the value of WIFSTOPPED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

WIFCONTINUED(*stat_val*)

Evaluates to a non-zero value if status was returned for a child process that has continued from a job control stop.

If the information pointed to by *stat_loc* was stored by a call to [waitpid\(\)](#) that specified the WUNTRACED flag and did not specify the WCONTINUED flag, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and WIFSTOPPED(**stat_loc*), will evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to [waitpid\(\)](#) that specified the WUNTRACED and WCONTINUED flags, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), WIFSTOPPED(**stat_loc*), and WIFCONTINUED(**stat_loc*), will evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to [waitpid\(\)](#) that did not specify the WUNTRACED or WCONTINUED flags, or by a call to the *wait()* function, exactly one of the macros WIFEXITED(**stat_loc*) and WIFSIGNALED(**stat_loc*) will evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to [waitpid\(\)](#) that did not specify the WUNTRACED flag and specified the WCONTINUED flag, or by a call to the *wait()* function, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and WIFCONTINUED(**stat_loc*), will evaluate to a non-zero value.

There may be additional implementation-dependent circumstances under which *wait()* or [waitpid\(\)](#) report status. This will not occur unless the calling process or one of its child processes explicitly makes use of a non-standard extension. In these cases the interpretation of the reported status is implementation-dependent.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes will be assigned a new parent process ID corresponding to an implementation-dependent system process.

RETURN VALUE

If *wait()* or [waitpid\(\)](#) returns because the status of a child process is available, these functions will return a value equal to the process ID of the child process for which status is reported. If *wait()* or [waitpid\(\)](#) returns due to the delivery of a signal to the calling process, -1 will be returned and *errno* will be set to [EINTR]. If [waitpid\(\)](#) was invoked with WNOHANG set in *options*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, 0 will be returned. Otherwise, (**pid_t**)-1 will be returned, and *errno* will be set to indicate the error.

ERRORS

The *wait()* function will fail if:

[ECHILD]

The calling process has no existing unwaited-for child processes.

[EINTR]

The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

The [waitpid\(\)](#) function will fail if:

[ECHILD]

The process or process group specified by *pid* does not exist or is not a child of the calling process.

[EINTR]

The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

[EINVAL]

The *options* argument is not valid.

EXAMPLES

None.

APPLICATION USAGE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[*exec*](#), [*exit\(\)*](#), [*fork\(\)*](#), [*wait3\(\)*](#), [*waitid\(\)*](#), [*<sys/types.h>*](#), [*<sys/wait.h>*](#).

DERIVATION

Derived from Issue 1 of the SVID.

UNIX ® is a registered Trademark of The Open Group.

Copyright © 1997 The Open Group

[[Main Index](#) | [XSH](#) | [XCU](#) | [XBD](#) | [XCURSES](#) | [XNS](#)]

NAME

waitpid - wait for a child process to stop or terminate

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

DESCRIPTION

Refer to [wait\(\)](#).

UNIX ® is a registered Trademark of The Open Group.
Copyright © 1997 The Open Group
[[Main Index](#) | [XSH](#) | [XCU](#) | [XBD](#) | [XCURSES](#) | [XNS](#)]



Search for

MSDN Library

Go

[Advanced Search](#)

sync toc

- Up One Level
- APCProc
- CancelWaitableTimer
- ChangeTimerQueueTimer
- CreateEvent
- CreateMutex
- CreateSemaphore
- CreateTimerQueue
- CreateTimerQueueTimer
- CreateWaitableTimer
- DeleteCriticalSection
- DeleteTimerQueue
- DeleteTimerQueueEx
- DeleteTimerQueueTimer
- EnterCriticalSection
- GetOverlappedResult
- InitializeCriticalSection
- InitializeCriticalSectionAndSpinCount
- InitializeSListHead
- InterlockedCompareExchange
- InterlockedCompareExchange64
- InterlockedCompareExchangeAcquire
- InterlockedCompareExchangeAcquire64
- InterlockedCompareExchangePointer
- InterlockedCompareExchangeRelease
- InterlockedCompareExchangeRelease64
- InterlockedDecrement
- InterlockedDecrement64
- InterlockedDecrementAcquire
- InterlockedDecrementRelease
- InterlockedExchange
- InterlockedExchange64
- InterlockedExchangeAcquire64
- InterlockedExchangeAdd
- InterlockedExchangeAdd64
- InterlockedExchangePointer
- InterlockedFlushSList

Welcome to the MSDN Library

[MSDN Home](#) > [MSDN Library](#) > [Windows Development](#) > [Windows Base Services](#) > [DLLs, Processes, and Threads](#) > [DLLs, Processes, and Threads](#) > [Synchronization](#) > [Synchronization Reference](#) > [Synchronization Functions](#)

Platform SDK: DLLs, Processes, and Threads

WaitForSingleObject

The **WaitForSingleObject** function returns when the specified object is in the signaled state or the time-out interval elapses.

To enter an alertable wait state, use the [WaitForSingleObjectEx](#) function. To wait for multiple objects, use the [WaitForMultipleObjects](#).

```
DWORD WaitForSingleObject(  
    HANDLE hHandle,  
    DWORD dwMilliseconds  
);
```

Parameters

hHandle
[in] Handle to the object. For a list of the object types whose handles can be specified, see the following Remarks section.

If this handle is closed while the wait is still pending, the function's behavior is undefined.

The handle must have the SYNCHRONIZE access right. For more information, see [Standard Access Rights](#).

dwMilliseconds
[in] Time-out interval, in milliseconds. The function returns if the interval elapses, even if the object's state is nonsignaled. If *dwMilliseconds* is zero, the function tests the object's state and returns immediately. If *dwMilliseconds* is INFINITE, the function's time-out interval never elapses.

Return Values

If the function succeeds, the return value indicates the event that caused the function to return. It can be one of the following values.

Return code	Description
WAIT_ABANDONED	The specified object is a mutex object that was not released by the thread that owned the mutex object before the owning thread terminated. Ownership of the mutex object is granted to the calling thread, and the mutex is set to nonsignaled.
WAIT_OBJECT_0	The state of the specified object is signaled.
WAIT_TIMEOUT	The time-out interval elapsed, and the object's state is nonsignaled.

If the function fails, the return value is WAIT_FAILED. To get extended error information, call [GetLastError](#).

Remarks

The **WaitForSingleObject** function checks the current state of the specified object. If the object's state is nonsignaled, the calling thread enters the wait state. It uses no processor time while waiting for the object state to become signaled or the time-out interval to elapse.

The function modifies the state of some types of synchronization objects. Modification occurs only for the object whose signaled state caused the function to return. For example, the

-  [InterlockedIncrement](#)
-  [InterlockedIncrement64](#)
-  [InterlockedIncrementAcquire](#)
-  [InterlockedIncrementRelease](#)
-  [InterlockedPopEntrySList](#)
-  [InterlockedPushEntrySList](#)
-  [LeaveCriticalSection](#)
-  [MsgWaitForMultipleObjects](#)
-  [MsgWaitForMultipleObjectsEx](#)
-  [OpenEvent](#)
-  [OpenMutex](#)
-  [OpenSemaphore](#)
-  [OpenWaitableTimer](#)
-  [PulseEvent](#)
-  [QueueUserAPC](#)
-  [RegisterWaitForSingleObject](#)
-  [ReleaseMutex](#)
-  [ReleaseSemaphore](#)
-  [ResetEvent](#)
-  [SetCriticalSectionSpinCount](#)
-  [SetEvent](#)
-  [SetWaitableTimer](#)
-  [SignalObjectAndWait](#)
-  [TimerAPCProc](#)
-  [TryEnterCriticalSection](#)
-  [UnregisterWait](#)
-  [UnregisterWaitEx](#)
-  [WaitForMultipleObjects](#)
-  [WaitForMultipleObjectsEx](#)
-  [WaitForSingleObject](#)
-  [WaitForSingleObjectEx](#)
-  [WaitOrTimerCallback](#)

count of a semaphore object is decreased by one.

The **WaitForSingleObject** function can wait for the following objects:

- Change notification
- Console input
- Event
- Job
- Memory resource notification
- Mutex
- Process
- Semaphore
- Thread
- Waitable timer

Use caution when calling the wait functions and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. A thread that uses a wait function with no time-out interval may cause the system to become deadlocked. Two examples of code that indirectly creates windows are DDE and the [CoInitialize](#) function. Therefore, if you have a thread that creates windows, use [MsgWaitForMultipleObjects](#) or [MsgWaitForMultipleObjectsEx](#), rather than **WaitForSingleObject**.

For more information, see [Using Mutex Objects](#).

Requirements

Client: Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.
Server: Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

[Synchronization Overview](#), [Synchronization Functions](#), [MsgWaitForMultipleObjects](#), [MsgWaitForMultipleObjectsEx](#), [WaitForMultipleObjects](#), [WaitForSingleObjectEx](#)

Last updated: May 2004 | [What did you think of this topic?](#) | [Order a Platform SDK CD](#)
 © 2004 Microsoft Corporation. All rights reserved. [Terms of use](#).

[Manage Your Profile](#) | [Legal](#) | [Contact Us](#) | [MSDN Flash Newsletter](#)

©2004 Microsoft Corporation. All rights reserved. [Terms of Use](#) | [Trademarks](#) | [Privacy Statement](#)





Search for

[Advanced Search](#)

Welcome to the MSDN Library

< meta name="MSHKeywordA" content="base.getexitcod

sync toc

- Up One Level
- AssignProcessToJobObject
- AttachThreadInput
- BindIoCompletionCallback
- CommandLineToArgvW
- ConvertFiberToThread
- ConvertThreadToFiber
- ConvertThreadToFiberEx
- CreateFiber
- CreateFiberEx
- CreateJobObject
- CreateProcess
- CreateProcessAsUser
- CreateProcessWithLogonW
- CreateProcessWithTokenW
- CreateRemoteThread
- CreateThread
- DeleteFiber
- ExitProcess
- ExitThread
- FiberProc
- FlsAlloc
- FlsCallback
- FlsFree
- FlsGetValue
- FlsSetValue
- FreeEnvironmentStrings
- GetCommandLine
- GetCurrentProcess
- GetCurrentProcessId
- GetCurrentThread
- GetCurrentThreadId
- GetEnvironmentStrings
- GetEnvironmentVariable

- ☐ GetExitCodeProcess
- ☐ GetExitCodeThread
- ☐ GetGuiResources
- ☐ GetCurrentProcessorNumber
- ☐ GetLogicalProcessorInformation
- ☐ GetNumaAvailableMemoryNode
- ☐ GetNumaHighestNodeNumber
- ☐ GetNumaNodeProcessorMask
- ☐ GetNumaProcessorNode
- ☐ GetPriorityClass
- ☐ GetProcessAffinityMask
- ☐ GetProcessHandleCount
- ☐ GetProcessId
- ☐ GetProcessIdOfThread
- ☐ GetProcessIoCounters
- ☐ GetProcessPriorityBoost
- ☐ GetProcessShutdownParameters
- ☐ GetProcessTimes
- ☐ GetProcessVersion
- ☐ GetProcessWorkingSetSize
- ☐ GetProcessWorkingSetSizeEx
- ☐ GetStartupInfo
- ☐ GetThreadId
- ☐ GetThreadIoPendingFlag
- ☐ GetThreadPriority
- ☐ GetThreadPriorityBoost
- ☐ GetThreadStartInformation
- ☐ GetThreadTimes
- ☐ IsProcessInJob
- ☐ NeedCurrentDirectoryForExePath
- ☐ NtQueryInformationProcess
- ☐ NtQueryInformationThread
- ☐ OpenJobObject
- ☐ OpenProcess
- ☐ OpenThread
- ☐ QueryInformationJobObject
- ☐ QueueUserWorkItem
- ☐ ResumeThread
- ☐ SetEnvironmentVariable
- ☐ SetInformationJobObject
- ☐ SetPriorityClass
- ☐ SetProcessAffinityMask
- ☐ SetProcessPriorityBoost
- ☐ SetProcessShutdownParameters
- ☐ SetProcessWorkingSetSize
- ☐ SetProcessWorkingSetSizeEx
- ☐ SetThreadAffinityMask
- ☐ SetThreadIdealProcessor

- ☐ SetThreadPriority
- ☐ SetThreadPriorityBoost
- ☐ Sleep
- ☐ SleepEx
- ☐ SuspendThread
- ☐ SwitchToFiber
- ☐ SwitchToThread
- ☐ TerminateJobObject
- ☐ TerminateProcess
- ☐ TerminateThread
- ☐ ThreadProc
- ☐ TlsAlloc
- ☐ TlsFree
- ☐ TlsGetValue
- ☐ TlsSetValue
- ☐ UserHandleGrantAccess
- ☐ WaitForInputIdle
- ☐ WinExec

NAME

environ, execl, execv, execl, execve, execlp, execvp - execute a file

SYNOPSIS

```
#include <unistd.h>
```

```
extern char **environ;  
int execl(const char *path, const char *arg0, ... /*,  
(char *)0 */);  
int execv(const char *path, char *const argv[]);  
int execl(const char *path,  
    const char *arg0, ... /*, (char *)0, char *const envp  
[] */);  
int execve(const char *path, char *const argv[], char  
*const envp[]);  
int execlp(const char *file, const char *arg0, ... /*,  
(char *)0 */);  
int execvp(const char *file, char *const argv[]);
```

DESCRIPTION

The *exec* functions replace the current process image with a new process image. The new image is constructed from a regular, executable file called the *new process image file*. There is no return from a successful *exec*, because the calling process image is overlaid by the new process image.

When a C-language program is executed as a result of this call, it is entered as a C-language function call as follows:

```
int main (int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialised as a pointer to an array of character pointers to the environment strings. The *argv* and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv* array is not counted in *argc*.

Conforming multi-threaded applications will not use the *environ* variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable is considered a use of the *environ* variable to access that environment variable.

The arguments specified by a program with one of the *exec* functions are passed on to the new process image in the corresponding *main()* arguments.

The argument *path* points to a pathname that identifies the new process image file.

The argument *file* is used to construct a pathname that identifies the new process image file. If the *file* argument contains a slash character, the *file* argument is used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable (see **XBD** specification, [Environment Variables](#)). If this environment variable is not present, the results of the search are implementation-dependent.

If the process image file is not a valid executable object, [execlp\(\)](#) and [execvp\(\)](#) use the contents of that file as standard input to a command interpreter conforming to [system\(\)](#). In this case, the command interpreter becomes the new process image.

The arguments represented by *arg0*, ... are pointers to null-terminated character strings. These strings constitute the argument list available to the new process image. The list is terminated by a null pointer. The argument *arg0* should point to a filename that is associated with the process being started by one of the *exec* functions.

The argument *argv* is an array of character pointers to null-terminated strings. The last member of this array must be a null pointer. These strings constitute the argument list available to the new process image. The value in *argv[0]* should point to a filename that is associated with the process being started by one of the *exec* functions.

The argument *envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The *envp* array is terminated by a null pointer.

For those forms not containing an *envp* pointer (i.e. *execl*, [execv\(\)](#), [execlp\(\)](#) and [execvp\(\)](#)), the environment for the new process image is taken from the external variable

environ in the calling process.

The number of bytes available for the new process' combined argument and environment lists is {ARG_MAX}. It is implementation-dependent whether null terminators, pointers, and/or any alignment bytes are included in this total.

File descriptors open in the calling process image remain open in the new process image, except for those whose close-on-exec flag FD_CLOEXEC is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged.

Directory streams open in the calling process image are closed in the new process image.

The state of conversion descriptors and message catalogue descriptors in the new process image is undefined. For the new process, the equivalent of:

```
setlocale(LC_ALL, "C")
```

is executed at startup.

Signals set to the default action (SIG_DFL) in the calling process image are set to the default action in the new process image. Signals set to be ignored (SIG_IGN) by the calling process image are set to be ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image (see [<signal.h>](#)). After a successful call to any of the *exec* functions, alternate signal stacks are not preserved and the SA_ONSTACK flag is cleared for all signals.

After a successful call to any of the *exec* functions, any functions previously registered by [atexit\(\)](#) are no longer registered.

If the ST_NOSUID bit is set for the file system containing the new process image file, then the effective user ID, effective group ID, saved set-user-ID and saved set-group-ID are unchanged in the new process image. Otherwise, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the new process image is set to the user ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved set-user-ID and the saved set-group-ID for use by [setuid\(\)](#)).

Any shared memory segments attached to the calling process image will not be attached to the new process image.

Any mappings established through [*mmap\(\)*](#) are not preserved across an *exec*.

If `_XOPEN_REALTIME` is defined and has a value other than -1, any named semaphores open in the calling process are closed as if by appropriate calls to [*sem_close\(\)*](#).

If the Process Memory Locking option is supported, memory locks established by the calling process via calls to [*mlockall\(\)*](#) or [*mlock\(\)*](#) are removed. If locked pages in the address space of the calling process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes will be unaffected by the call by this process to the *exec* function. If the *exec* function fails, the effect on memory locks is unspecified.

Memory mappings created in the process are unmapped before the address space is rebuilt for the new process image.

If the Process Scheduling option is supported, for the `SCHED_FIFO` and `SCHED_RR` scheduling policies, the policy and priority settings are not changed by a call to an *exec* function. For other scheduling policies, the policy and priority settings on *exec* are implementation-dependent.

If the Timers option is supported, per-process timers created by the calling process are deleted before replacing the current process image with the new process image.

If the Message Passing option is supported, all open message queue descriptors in the calling process are closed, as described in [*mq_close\(\)*](#).

If the Asynchronous Input and Output option is supported, any outstanding asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled will complete as if the *exec* function had not yet occurred, but any associated signal notifications are suppressed. It is unspecified whether the *exec* function itself blocks awaiting such I/O completion. In no event, however, will the new process image created by the *exec* function be affected by the presence of outstanding asynchronous I/O operations at the time the *exec* function is called. Whether any I/O is cancelled, and which I/O may be cancelled upon *exec*, is implementation-dependent.

The new process also inherits at least the following attributes from the calling process image:

nice value (see [*nice\(\)*](#))

semadj values (see [*semop\(\)*](#))

process ID
 parent process ID
 process group ID
 session membership
 real user ID
 real group ID
 supplementary group IDs
 time left until an alarm clock signal (see [alarm\(\)](#))
 current working directory
 root directory
 file mode creation mask (see [umask\(\)](#))
 file size limit (see [ulimit\(\)](#))
 process signal mask (see [sigprocmask\(\)](#))
 pending signal (see [sigpending\(\)](#))
tms_utime, *tms_stime*, *tms_cutime*, and *tms_cstime* (see [times\(\)](#))
 resource limits
 controlling terminal
 interval timers

All other process attributes defined in this document will be the same in the new and old process images. The inheritance of process attributes not defined by this specification is implementation-dependent.

A call to any *exec* function from a process with more than one thread results in all threads being terminated and the new executable image being loaded and executed. No destructor functions will be called.

Upon successful completion, the *exec* functions mark for update the *st_atime* field of the file. If an *exec* function failed but was able to locate the *process image file*, whether the *st_atime* field is marked for update is unspecified. Should the *exec* function succeed, the process image file is considered to have been opened with [open\(\)](#). The corresponding [close\(\)](#) is considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the *exec* functions. The *argv[]* and *envp[]* arrays of pointers and the strings to which those arrays point will not be modified by a call to one of the *exec* functions, except as a consequence of replacing the process image.

The saved resource limits in the new process image are set to be a copy of the process's corresponding hard and soft limits.

RETURN VALUE

If one of the *exec* functions returns to the calling process image, an error has occurred; the return value is -1, and *errno* is set to indicate the error.

ERRORS

The *exec* functions will fail if:

[E2BIG]

The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit of {ARG_MAX} bytes.

[EACCES]

Search permission is denied for a directory listed in the new process image file's path prefix, or the new process image file denies execution permission, or the new process image file is not a regular file and the implementation does not support execution of files of its type.

[ELOOP]

Too many symbolic links were encountered in resolving *path*.

[ENAMETOOLONG]

The length of the *path* or *file* arguments, or an element of the environment variable prefixed to a file, exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX}.

[ENOENT]

A component of *path* or *file* does not name an existing file or *path* or *file* is an empty string.

[ENOTDIR]

A component of the new process image file's path prefix is not a directory.

The *exec* functions, except for [*execlp\(\)*](#) and [*execvp\(\)*](#), will fail if:

[ENOEXEC]

The new process image file has the appropriate access permission but is not in the proper format.

The *exec* functions may fail if:

[ENAMETOOLONG]

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

[ENOMEM]

The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.

[ETXTBSY]

The new process image file is a pure procedure (shared text) file that is currently open for writing by some process.

EXAMPLES

None.

APPLICATION USAGE

As the state of conversion descriptors and message catalogue descriptors in the new process image is undefined, portable applications should not rely on their use and should close them prior to calling one of the *exec* functions.

Applications that require other than the default POSIX locale should call [*setlocale\(\)*](#) with the appropriate parameters to establish the locale of the new process.

The *environ* array should not be accessed directly by the application.

FUTURE DIRECTIONS

None.

SEE ALSO

[*alarm\(\)*](#), [*atexit\(\)*](#), [*chmod\(\)*](#), [*exit\(\)*](#), [*fcntl\(\)*](#), [*fork\(\)*](#), [*fstatvfs\(\)*](#), [*getenv\(\)*](#), [*getitimer\(\)*](#), [*getrlimit\(\)*](#), [*mmap\(\)*](#), [*nice\(\)*](#), [*putenv\(\)*](#), [*semop\(\)*](#), [*setlocale\(\)*](#), [*shmat\(\)*](#), [*sigaction\(\)*](#), [*sigaltstack\(\)*](#), [*sigpending\(\)*](#), [*sigprocmask\(\)*](#), [*system\(\)*](#), [*times\(\)*](#), [*ulimit\(\)*](#), [*umask\(\)*](#), [*<unistd.h>*](#), **XBD** specification, [**General Terminal Interface**](#).

DERIVATION

Derived from Issue 1 of the SVID.

UNIX ® is a registered Trademark of The Open Group.

Copyright © 1997 The Open Group

[[Main Index](#) | [XSH](#) | [XCU](#) | [XBD](#) | [XCURSES](#) | [XNS](#)]

NAME

vfork - create new process; share virtual memory

SYNOPSIS

```
#include <unistd.h>
```

```
pid_t vfork(void);
```

DESCRIPTION

The *vfork()* function has the same effect as [fork\(\)](#), except that the behaviour is undefined if the process created by *vfork()* either modifies any data other than a variable of type **pid_t** used to store the return value from *vfork()*, or returns from the function in which *vfork()* was called, or calls any other function before successfully calling [_exit\(\)](#) or one of the *exec* family of functions.

RETURN VALUE

Upon successful completion, *vfork()* returns 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, -1 is returned to the parent, no child process is created, and *errno* is set to indicate the error.

ERRORS

The *vfork()* function will fail if:

[EAGAIN]

The system-wide limit on the total number of processes under execution would be exceeded, or the system-imposed limit on the total number of processes under execution by a single user would be exceeded.

[ENOMEM]

There is insufficient swap space for the new process.

EXAMPLES

None.

APPLICATION USAGE

On some systems, *vfork()* is the same as [fork\(\)](#).

The *vfork()* function differs from [fork\(\)](#) only in that the child process can share code and data with the calling process (parent process). This speeds cloning activity significantly at a risk to the integrity of the parent process if *vfork()* is misused.

The use of *vfork()* for any purpose except as a prelude to an immediate call to a function from the *exec* family, or to [_exit\(\)](#), is not advised.

The *vfork()* function can be used to create new processes without fully copying the address space of the old process. If a forked process is simply going to call *exec*, the data space copied from the parent to the child by [fork\(\)](#) is not used. This is particularly inefficient in a paged environment, making *vfork()* particularly useful. Depending upon the size of the parent's data space, *vfork()* can give a significant performance improvement over [fork\(\)](#).

The *vfork()* function can normally be used just like [fork\(\)](#). It does not work, however, to return while running in the child's context from the caller of *vfork()* since the eventual return from *vfork()* would then return to a no longer existent stack frame. Be careful, also, to call [_exit\(\)](#) rather than [exit\(\)](#) if you cannot *exec*, since [exit\(\)](#) flushes and closes standard I/O channels, thereby damaging the parent process' standard I/O data structures. (Even with [fork\(\)](#), it is wrong to call [exit\(\)](#), since buffered data would then be flushed twice.)

If signal handlers are invoked in the child process after *vfork()*, they must follow the same rules as other code in the child process.

FUTURE DIRECTIONS

None.

SEE ALSO

[exec](#), [exit\(\)](#), [fork\(\)](#), [wait\(\)](#), [<unistd.h>](#).

UNIX ® is a registered Trademark of The Open Group.

Copyright © 1997 The Open Group

[[Main Index](#) | [XSH](#) | [XCU](#) | [XBD](#) | [XCURSES](#) | [XNS](#)]

```
#include <stdlib.h>

int main (int argc, char** argv) {
    printf("parent\n");
    int i, pid;
    int execResult;

    if ((pid=vfork()) == -1) {
        printf("error creating process\n");
        exit(1);
    }

    if (pid == 0) {
        execResult = execlp("./child.exe", "child", NULL);
        //only gets here in case of an error
        printf("execlp returned: %d\n",execResult);
    }
    if (pid > 0) {
        printf("child's pid=%d\n",pid);
        for (i=0; i<10; i++) {
            sleep(1);
            printf("parent again %d\n",i);
        }
    }
    return 0;
}
```

```
int main(int argc, char** argv) {  
    int i;  
  
    for (i=0; i<10; i++) {  
        printf("child %d\n",i);  
        sleep(1);  
    }  
}
```



Search for

MSDN Library

Go

Advanced Search

sync toc ↻ x

- Up One Level
- AssignProcessToJobObject
- AttachThreadInput
- BindIoCompletionCallback
- CommandLineToArgvW
- ConvertFiberToThread
- ConvertThreadToFiber
- ConvertThreadToFiberEx
- CreateFiber
- CreateFiberEx
- CreateJobObject
- CreateProcess
- CreateProcessAsUser
- CreateProcessWithLogonW
- CreateProcessWithTokenW
- CreateRemoteThread
- CreateThread
- DeleteFiber
- ExitProcess
- ExitThread
- FiberProc
- FlsAlloc
- FlsCallback
- FlsFree
- FlsGetValue
- FlsSetValue
- FreeEnvironmentStrings
- GetCommandLine
- GetCurrentProcess
- GetCurrentProcessId
- GetCurrentThread
- GetCurrentThreadId
- GetEnvironmentStrings
- GetEnvironmentVariable

Welcome to the MSDN Library

[MSDN Home](#) > [MSDN Library](#) > [Windows Development](#) > [Windows Base Services](#) > [DLLs, Processes, and Threads](#) > [DLLs, Processes, and Threads](#) > [Processes and Threads](#) > [Process and Thread Reference](#) > [Process and Thread Functions](#)

Platform SDK: DLLs, Processes, and Threads

CreateProcess

The **CreateProcess** function creates a new process and its primary thread. The new process runs the specified executable file in the security context of the calling process.

If the calling process is impersonating another user, the new process uses the token for the calling process, not the impersonation token. To run the new process in the security context of the user represented by the impersonation token, use the [CreateProcessAsUser](#) or [CreateProcessWithLogonW](#) function.

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,
    LPCTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

Parameters

lpApplicationName
[in] Pointer to a null-terminated string that specifies the module to execute. The specified module can be a Windows-based application. It can be some other type of module (for example, MS-DOS or OS/2) if the appropriate subsystem is available on the local computer.

The string can specify the full path and file name of the module to execute or it can specify a partial name. In the case of a partial name, the function uses the current drive and current directory to complete the specification. The function will not use the search path. If the file name does not contain an extension, .exe is assumed. Therefore, if the file name extension is .com, this parameter must include the .com extension.

The *lpApplicationName* parameter can be NULL. In that case, the module name must be the first white space-delimited token in the *lpCommandLine* string. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin; otherwise, the file name is ambiguous. For example, consider the string "c:\program files\sub dir\program name". This string can be interpreted in a number of ways. The system tries to interpret the possibilities in the following order:

```
c:\program.exe files\sub dir\program name
c:\program files\sub.exe dir\program name
c:\program files\sub dir\program.exe name
c:\program files\sub dir\program name.exe
```

If the executable module is a 16-bit application, *lpApplicationName* should be NULL, and the string pointed to by *lpCommandLine* should specify the executable module as well as its arguments.

lpCommandLine
[in, out] Pointer to a null-terminated string that specifies the command line to execute. The maximum length of this string is 32K characters.

- ☐ GetExitCodeProcess
- ☐ GetExitCodeThread
- ☐ GetGuiResources
- ☐ GetCurrentProcessorNumber
- ☐ GetLogicalProcessorInformation
- ☐ GetNumaAvailableMemoryNode
- ☐ GetNumaHighestNodeNumber
- ☐ GetNumaNodeProcessorMask
- ☐ GetNumaProcessorNode
- ☐ GetPriorityClass
- ☐ GetProcessAffinityMask
- ☐ GetProcessHandleCount
- ☐ GetProcessId
- ☐ GetProcessIdOfThread
- ☐ GetProcessIoCounters
- ☐ GetProcessPriorityBoost
- ☐ GetProcessShutdownParameters
- ☐ GetProcessTimes
- ☐ GetProcessVersion
- ☐ GetProcessWorkingSetSize
- ☐ GetProcessWorkingSetSizeEx
- ☐ GetStartupInfo
- ☐ GetThreadId
- ☐ GetThreadIoPendingFlag
- ☐ GetThreadPriority
- ☐ GetThreadPriorityBoost
- ☐ GetThreadStartInformation
- ☐ GetThreadTimes
- ☐ IsProcessInJob
- ☐ NeedCurrentDirectoryForExePath
- ☐ NtQueryInformationProcess
- ☐ NtQueryInformationThread
- ☐ OpenJobObject
- ☐ OpenProcess
- ☐ OpenThread
- ☐ QueryInformationJobObject
- ☐ QueueUserWorkItem
- ☐ ResumeThread
- ☐ SetEnvironmentVariable
- ☐ SetInformationJobObject
- ☐ SetPriorityClass
- ☐ SetProcessAffinityMask
- ☐ SetProcessPriorityBoost
- ☐ SetProcessShutdownParameters
- ☐ SetProcessWorkingSetSize
- ☐ SetProcessWorkingSetSizeEx
- ☐ SetThreadAffinityMask
- ☐ SetThreadIdealProcessor

Windows 2000: The maximum length of this string is MAX_PATH characters.

The Unicode version of this function, **CreateProcessW**, will fail if this parameter is a **const** string.

The *lpCommandLine* parameter can be NULL. In that case, the function uses the string pointed to by *lpApplicationName* as the command line.

If both *lpApplicationName* and *lpCommandLine* are non-NULL, the null-terminated string pointed to by *lpApplicationName* specifies the module to execute, and the null-terminated string pointed to by *lpCommandLine* specifies the command line. The new process can use [GetCommandLine](#) to retrieve the entire command line. Console processes written in C can use the *argc* and *argv* arguments to parse the command line. Because *argv[0]* is the module name, C programmers generally repeat the module name as the first token in the command line.

If *lpApplicationName* is NULL, the first white-space – delimited token of the command line specifies the module name. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin (see the explanation for the *lpApplicationName* parameter). If the file name does not contain an extension, .exe is appended. Therefore, if the file name extension is .com, this parameter must include the .com extension. If the file name ends in a period (.) with no extension, or if the file name contains a path, .exe is not appended. If the file name does not contain a directory path, the system searches for the executable file in the following sequence:

1. The directory from which the application loaded.
2. The current directory for the parent process.
3. The 32-bit Windows system directory. Use the [GetSystemDirectory](#) function to get the path of this directory.

Windows Me/98/95: The Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory.

4. The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched. The name of this directory is System.
5. The Windows directory. Use the [GetWindowsDirectory](#) function to get the path of this directory.
6. The directories that are listed in the PATH environment variable.

The system adds a null character to the command line string to separate the file name from the arguments. This divides the original string into two strings for internal processing.

lpProcessAttributes

[in] Pointer to a [SECURITY_ATTRIBUTES](#) structure that determines whether the returned handle can be inherited by child processes. If *lpProcessAttributes* is NULL, the handle cannot be inherited.

The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new process. If *lpProcessAttributes* is NULL or **lpSecurityDescriptor** is NULL, the process gets a default security descriptor. The ACLs in the default security descriptor for a process come from the primary or impersonation token of the creator.

lpThreadAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. If *lpThreadAttributes* is NULL, the handle cannot be inherited.

The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the main thread. If *lpThreadAttributes* is NULL or **lpSecurityDescriptor** is NULL, the thread gets a default security descriptor. The ACLs in the default security descriptor for a thread come from the primary or impersonation token of the creator.

bInheritHandles

[in] If this parameter TRUE, each inheritable handle in the calling process is inherited by the new process. If the parameter is FALSE, the handles are not inherited. Note that inherited handles have the same value and access rights as the original handles.

dwCreationFlags

[in] Flags that control the priority class and the creation of the process. For a list of values, see [Process Creation Flags](#).

This parameter also controls the new process's priority class, which is used to determine the scheduling priorities of the process's threads. For a list of values, see [GetPriorityClass](#). If none of the priority class flags is specified, the priority class defaults to NORMAL_PRIORITY_CLASS unless the priority class of the creating process is IDLE_PRIORITY_CLASS or BELOW_NORMAL_PRIORITY_CLASS. In this case, the child process receives the default priority class of the calling process.

- ☐ [SetThreadPriority](#)
- ☐ [SetThreadPriorityBoost](#)
- ☐ [Sleep](#)
- ☐ [SleepEx](#)
- ☐ [SuspendThread](#)
- ☐ [SwitchToFiber](#)
- ☐ [SwitchToThread](#)
- ☐ [TerminateJobObject](#)
- ☐ [TerminateProcess](#)
- ☐ [TerminateThread](#)
- ☐ [ThreadProc](#)
- ☐ [TlsAlloc](#)
- ☐ [TlsFree](#)
- ☐ [TlsGetValue](#)
- ☐ [TlsSetValue](#)
- ☐ [UserHandleGrantAccess](#)
- ☐ [WaitForInputIdle](#)
- ☐ [WinExec](#)

lpEnvironment

[in] Pointer to an environment block for the new process. If this parameter is NULL, the new process uses the environment of the calling process.

An environment block consists of a null-terminated block of null-terminated strings. Each string is in the form:

name=value

Because the equal sign is used as a separator, it must not be used in the name of an environment variable.

An environment block can contain either Unicode or ANSI characters. If the environment block pointed to by *lpEnvironment* contains Unicode characters, be sure that *dwCreationFlags* includes CREATE_UNICODE_ENVIRONMENT.

Note that an ANSI environment block is terminated by two zero bytes: one for the last string, one more to terminate the block. A Unicode environment block is terminated by four zero bytes: two for the last string, two more to terminate the block.

lpCurrentDirectory

[in] Pointer to a null-terminated string that specifies the full path to the current directory for the process. The string can also specify a UNC path.

If this parameter is NULL, the new process will have the same current drive and directory as the calling process. (This feature is provided primarily for shells that need to start an application and specify its initial drive and working directory.)

lpStartupInfo

[in] Pointer to a [STARTUPINFO](#) structure that specifies the window station, desktop, standard handles, and appearance of the main window for the new process.

lpProcessInformation

[out] Pointer to a [PROCESS_INFORMATION](#) structure that receives identification information about the new process.

Handles in **PROCESS_INFORMATION** must be closed with [CloseHandle](#) when they are no longer needed.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The process is assigned a process identifier. The identifier is valid until the process terminates. It can be used to identify the process, or specified in the [OpenProcess](#) function to open a handle to the process. The initial thread in the process is also assigned a thread identifier. It can be specified in the [OpenThread](#) function to open a handle to the thread. The identifier is valid until the thread terminates and can be used to uniquely identify the thread within the system. These identifiers are returned in the [PROCESS_INFORMATION](#) structure.

The calling thread can use the [WaitForInputIdle](#) function to wait until the new process has finished its initialization and is waiting for user input with no input pending. This can be useful for synchronization between parent and child processes, because **CreateProcess** returns without waiting for the new process to finish its initialization. For example, the creating process would use **WaitForInputIdle** before trying to find a window associated with the new process.

The preferred way to shut down a process is by using the [ExitProcess](#) function, because this function sends notification of approaching termination to all DLLs attached to the process. Other means of shutting down a process do not notify the attached DLLs. Note that when a thread calls **ExitProcess**, other threads of the process are terminated without an opportunity to execute any additional code (including the thread termination code of attached DLLs). For

more information, see [Terminating a Process](#).

A parent process can directly alter the environment variables of a child process during process creation. This is the only situation when a process can directly change the environment settings of another process. For more information, see [Changing Environment Variables](#).

If an application provides an environment block, the current directory information of the system drives is not automatically propagated to the new process. For example, there is an environment variable named `=C:` whose value is the current directory on drive C. An application must manually pass the current directory information to the new process. To do so, the application must explicitly create these environment variable strings, sort them alphabetically (because the system uses a sorted environment), and put them into the environment block. Typically, they will go at the front of the environment block, due to the environment block sort order.

One way to obtain the current directory information for a drive X is to call [GetFullPathName](#) ("X:;. ."). That avoids an application having to scan the environment block. If the full path returned is X:\, there is no need to pass that value on as environment data, since the root directory is the default current directory for drive X of a new process.

Note The name of the executable in the command line that the operating system provides to a process is not necessarily identical to that in the command line that the calling process gives to the **CreateProcess** function. The operating system may prepend a fully qualified path to an executable name that is provided without a fully qualified path.

When a process is created with `CREATE_NEW_PROCESS_GROUP` specified, an implicit call to [SetConsoleCtrlHandler](#)(NULL,TRUE) is made on behalf of the new process; this means that the new process has CTRL+C disabled. This lets shells handle CTRL+C themselves, and selectively pass that signal on to sub-processes. CTRL+BREAK is not disabled, and may be used to interrupt the process/process group.

Windows Me/98/95: **CreateProcessW** is supported by the Microsoft Layer for Unicode. To use this, you must add certain files to your application, as outlined in [Microsoft Layer for Unicode on Windows Me/98/95 Systems](#).

Security Remarks

The first parameter, *lpApplicationName*, can be NULL, in which case the executable name must be in the white space-delimited string pointed to by *lpCommandLine*. If the executable or path name has a space in it, there is a risk that a different executable could be run because of the way the function parses spaces. The following example is dangerous because the function will attempt to run "Program.exe", if it exists, instead of "MyApp.exe".

```
CreateProcess(NULL, "C:\\Program Files\\MyApp", ...)
```

If a malicious user were to create an application called "Program.exe" on a system, any program that incorrectly calls **CreateProcess** using the Program Files directory will run this application instead of the intended application.

To avoid this problem, do not pass NULL for *lpApplicationName*. If you do pass NULL for *lpApplicationName*, use quotation marks around the executable path in *lpCommandLine*, as shown in the example below.

```
CreateProcess(NULL, "\"C:\\Program Files\\MyApp.exe\" -L -S", ...)
```

Example Code

For an example, see [Creating Processes](#).

Requirements

Client: Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.

Server: Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.

Unicode: Implemented as Unicode and ANSI versions. Note that Unicode support on Windows Me/98/95 requires Microsoft Layer for Unicode.

Header: Declared in Winbase.h; include Windows.h.

Library: Use Kernel32.lib.

See Also

[Processes and Threads Overview](#), [Process and Thread Functions](#), [CloseHandle](#), [CreateProcessAsUser](#), [CreateProcessWithLogonW](#), [ExitProcess](#), [GetCommandLine](#), [GetEnvironmentStrings](#), [GetExitCodeProcess](#), [GetFullPathName](#), [GetStartupInfo](#), [OpenProcess](#), [PROCESS_INFORMATION](#), [SECURITY_ATTRIBUTES](#), [SetErrorMode](#), [STARTUPINFO](#), [TerminateProcess](#), [WaitForInputIdle](#)

Last updated: May 2004 | [What did you think of this topic?](#) | [Order a Platform SDK CD](#)
[© 2004 Microsoft Corporation. All rights reserved. Terms of use.](#)

[Manage Your Profile](#) | [Legal](#) | [Contact Us](#) | [MSDN Flash Newsletter](#)

©2004 Microsoft Corporation. All rights reserved. [Terms of Use](#) | [Trademarks](#) | [Privacy Statement](#)




```
#include <Windows.h>
```

```
int main( VOID ) {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    LPDWORD result;
    int i;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    // Start the child process.
    if( !CreateProcess( NULL, // No module name (use command line).
        "childWin.exe", // Command line.
        NULL,           // Process handle not inheritable.
        NULL,           // Thread handle not inheritable.
        FALSE,          // Set handle inheritance to FALSE.
        0,              // No creation flags.
        NULL,           // Use parent's environment block.
        NULL,           // Use parent's starting directory.
        &si,             // Pointer to STARTUPINFO structure.
        &pi )           // Pointer to PROCESS_INFORMATION structure.
    ) {
        return 1;
    }

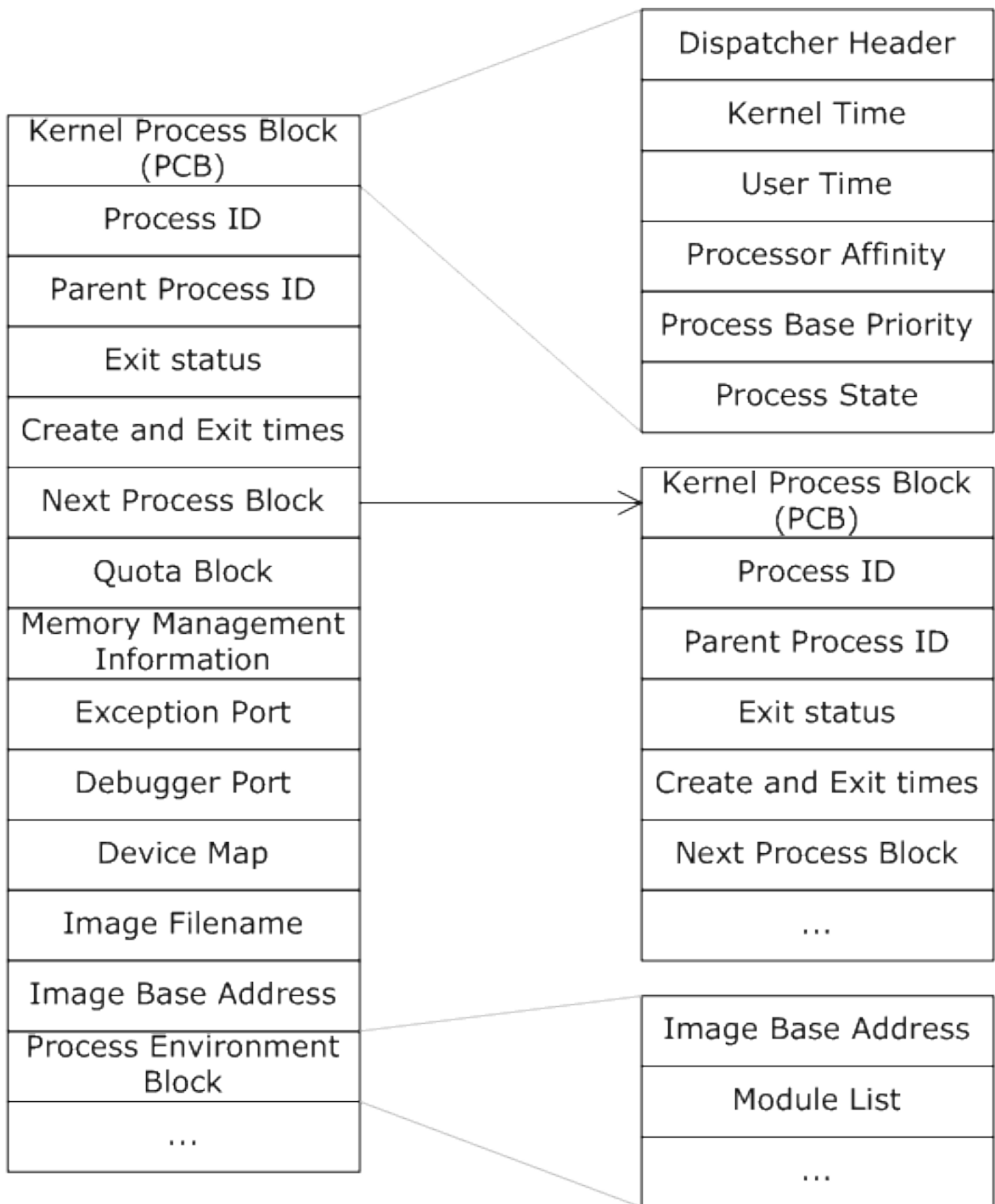
    for (i=0; i<10;i++) {
        printf("parent %d\n",i);
        Sleep(1000);
    }
    //wait for child to exit
    WaitForSingleObject(pi.hProcess, INFINITE);
    GetExitCodeProcess(pi.hProcess, result);
    printf("child's exit code was: %i\n", *result);

    return 0;
}
```

parent 0
child 0
parent 1
child 1
parent 2
child 2
parent 3
child 3
parent 4
child 4
parent 5
child 5
parent 6
child 6
parent 7
child 7
parent 8
child 8
parent 9
child 9

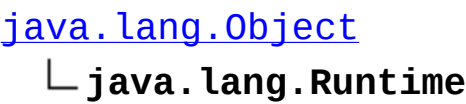
```
#include <Windows.h>

int main(int argc, char** argv) {
    int i;
    for (i=0; i<10; i++) {
        printf("child %d\n",i);
        Sleep(1000);
    }
    exit(2);
}
```



java.lang

Class Runtime



public class **Runtime**

extends [Object](#)

Every Java application has a single instance of class `Runtime` that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the `getRuntime` method.

An application cannot create its own instance of this class.

Since: JDK1.0

See Also: [getRuntime\(\)](#)

Method Summary	
void	addShutdownHook (Thread hook) Registers a new virtual-machine shutdown hook.
int	availableProcessors () Returns the number of processors available to the Java virtual machine.
Process	exec (String command) Executes the specified string command in a separate process.
Process	exec (String [] cmdarray) Executes the specified command and arguments in a separate process.

Process	exec (String [] cmdarray, String [] envp) Executes the specified command and arguments in a separate process with the specified environment.
Process	exec (String [] cmdarray, String [] envp, File dir) Executes the specified command and arguments in a separate process with the specified environment and working directory.
Process	exec (String command, String [] envp) Executes the specified string command in a separate process with the specified environment.
Process	exec (String command, String [] envp, File dir) Executes the specified string command in a separate process with the specified environment and working directory.
void	exit (int status) Terminates the currently running Java virtual machine by initiating its shutdown sequence.
long	freeMemory () Returns the amount of free memory in the Java Virtual Machine.
void	gc () Runs the garbage collector.
InputStream	getLocalizedInputStream (InputStream in) Deprecated. <i>As of JDK 1.1, the preferred way to translate a byte stream in the local encoding into a character stream in Unicode is via the InputStreamReader and BufferedReader classes.</i>
OutputStream	getLocalizedOutputStream (OutputStream out) Deprecated. <i>As of JDK 1.1, the preferred way to translate a Unicode character stream into a byte stream in the local encoding is via the OutputStreamWriter, BufferedWriter, and PrintWriter classes.</i>
static Runtime	getRuntime () Returns the runtime object associated with the current Java application.
void	halt (int status) Forcibly terminates the currently running Java virtual machine.
void	load (String filename) Loads the specified filename as a dynamic library.
void	loadLibrary (String libname) Loads the dynamic library with the specified library name.
long	maxMemory () Returns the maximum amount of memory that the Java virtual machine will attempt to use.

boolean	<code>removeShutdownHook</code> (<code>Thread</code> hook) De-registers a previously-registered virtual-machine shutdown hook.
void	<code>runFinalization</code> () Runs the finalization methods of any objects pending finalization.
static void	<code>runFinalizersOnExit</code> (boolean value) Deprecated. <i>This method is inherently unsafe. It may result in finalizers being called on live objects while other threads are concurrently manipulating those objects, resulting in erratic behavior or deadlock.</i>
long	<code>totalMemory</code> () Returns the total amount of memory in the Java virtual machine.
void	<code>traceInstructions</code> (boolean on) Enables/Disables tracing of instructions.
void	<code>traceMethodCalls</code> (boolean on) Enables/Disables tracing of method calls.

Methods inherited from class [java.lang.Object](#)

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

Method Detail

getRuntime

```
public static Runtime getRuntime()
```

Returns the runtime object associated with the current Java application. Most of the methods of class `Runtime` are instance methods and must be invoked with respect to the current runtime object.

Returns:

the `Runtime` object associated with the current Java application.

exit

```
public void exit(int status)
```

Terminates the currently running Java virtual machine by initiating its shutdown sequence. This method never returns normally. The argument serves as a status code; by convention, a nonzero status code indicates abnormal termination.

The virtual machine's shutdown sequence consists of two phases. In the first phase all registered [shutdown hooks](#), if any, are started in some unspecified order and allowed to run concurrently until they finish. In the second phase all uninvoked finalizers are run if [finalization-on-exit](#) has been enabled. Once this is done the virtual machine [halts](#).

If this method is invoked after the virtual machine has begun its shutdown sequence then if shutdown hooks are being run this method will block indefinitely. If shutdown hooks have already been run and on-exit finalization has been enabled then this method halts the virtual machine with the given status code if the status is nonzero; otherwise, it blocks indefinitely.

The [System.exit](#) method is the conventional and convenient means of invoking this method.

Parameters:

`status` - Termination status. By convention, a nonzero status code indicates abnormal termination.

Throws:

[SecurityException](#) - If a security manager is present and its [checkExit](#) method does not permit exiting with the specified status

See Also:

[SecurityException](#), [SecurityManager.checkExit\(int\)](#), [addShutdownHook\(java.lang.Thread\)](#), [removeShutdownHook\(java.lang.Thread\)](#), [runFinalizersOnExit\(boolean\)](#), [halt\(int\)](#)

addShutdownHook

```
public void addShutdownHook(Thread hook)
```

Registers a new virtual-machine shutdown hook.

The Java virtual machine *shuts down* in response to two kinds of events:

- The program *exits* normally, when the last non-daemon thread exits or when the [exit](#) (equivalently, [System.exit](#)) method is invoked, or
- The virtual machine is *terminated* in response to a user interrupt, such as typing ^C, or

a system-wide event, such as user logoff or system shutdown.

A *shutdown hook* is simply an initialized but unstarted thread. When the virtual machine begins its shutdown sequence it will start all registered shutdown hooks in some unspecified order and let them run concurrently. When all the hooks have finished it will then run all uninvoked finalizers if finalization-on-exit has been enabled. Finally, the virtual machine will halt. Note that daemon threads will continue to run during the shutdown sequence, as will non-daemon threads if shutdown was initiated by invoking the [exit](#) method.

Once the shutdown sequence has begun it can be stopped only by invoking the [halt](#) method, which forcibly terminates the virtual machine.

Once the shutdown sequence has begun it is impossible to register a new shutdown hook or de-register a previously-registered hook. Attempting either of these operations will cause an [IllegalStateException](#) to be thrown.

Shutdown hooks run at a delicate time in the life cycle of a virtual machine and should therefore be coded defensively. They should, in particular, be written to be thread-safe and to avoid deadlocks insofar as possible. They should also not rely blindly upon services that may have registered their own shutdown hooks and therefore may themselves in the process of shutting down.

Shutdown hooks should also finish their work quickly. When a program invokes [exit](#) the expectation is that the virtual machine will promptly shut down and exit. When the virtual machine is terminated due to user logoff or system shutdown the underlying operating system may only allow a fixed amount of time in which to shut down and exit. It is therefore inadvisable to attempt any user interaction or to perform a long-running computation in a shutdown hook.

Uncaught exceptions are handled in shutdown hooks just as in any other thread, by invoking the [uncaughtException](#) method of the thread's [ThreadGroup](#) object. The default implementation of this method prints the exception's stack trace to [System.err](#) and terminates the thread; it does not cause the virtual machine to exit or halt.

In rare circumstances the virtual machine may *abort*, that is, stop running without shutting down cleanly. This occurs when the virtual machine is terminated externally, for example with the SIGKILL signal on Unix or the `TerminateProcess` call on Microsoft Windows. The virtual machine may also abort if a native method goes awry by, for example, corrupting internal data structures or attempting to access nonexistent memory. If the virtual machine aborts then no guarantee can be made about whether or not any shutdown hooks will be run.

Parameters:

hook - An initialized but unstarted [Thread](#) object

Throws:

[IllegalArgumentException](#) - If the specified hook has already been registered, or if it can be determined that the hook is already running or has already been run
[IllegalStateException](#) - If the virtual machine is already in the process of shutting down
[SecurityException](#) - If a security manager is present and it denies
[RuntimePermission](#)("shutdownHooks")

Since:

1.3

See Also:

[removeShutdownHook\(java.lang.Thread\)](#), [halt\(int\)](#), [exit\(int\)](#)

removeShutdownHook

```
public boolean removeShutdownHook(Thread hook)
```

De-registers a previously-registered virtual-machine shutdown hook.

Parameters:

hook - the hook to remove

Returns:

true if the specified hook had previously been registered and was successfully de-registered, false otherwise.

Throws:

[IllegalStateException](#) - If the virtual machine is already in the process of shutting down
[SecurityException](#) - If a security manager is present and it denies
[RuntimePermission](#)("shutdownHooks")

Since:

1.3

See Also:

[addShutdownHook\(java.lang.Thread\)](#), [exit\(int\)](#)

halt

```
public void halt(int status)
```

Forcibly terminates the currently running Java virtual machine. This method never returns normally.

This method should be used with extreme caution. Unlike the [exit](#) method, this method does not cause shutdown hooks to be started and does not run uninvoked finalizers if finalization-on-exit has been enabled. If the shutdown sequence has already been initiated then this method does not wait for any running shutdown hooks or finalizers to finish their work.

Parameters:

`status` - Termination status. By convention, a nonzero status code indicates abnormal termination. If the [exit](#) (equivalently, [System.exit](#)) method has already been invoked then this status code will override the status code passed to that method.

Throws:

[SecurityException](#) - If a security manager is present and its [checkExit](#) method does not permit an exit with the specified status

Since:

1.3

See Also:

[exit\(int\)](#), [addShutdownHook\(java.lang.Thread\)](#),
[removeShutdownHook\(java.lang.Thread\)](#)

runFinalizersOnExit

```
public static void runFinalizersOnExit(boolean value)
```

Deprecated. *This method is inherently unsafe. It may result in finalizers being called on live objects while other threads are concurrently manipulating those objects, resulting in erratic behavior or deadlock.*

Enable or disable finalization on exit; doing so specifies that the finalizers of all objects that have finalizers that have not yet been automatically invoked are to be run before the Java runtime exits. By default, finalization on exit is disabled.

If there is a security manager, its `checkExit` method is first called with 0 as its argument to ensure the exit is allowed. This could result in a `SecurityException`.

Parameters:

`value` - true to enable finalization on exit, false to disable

Throws:

[SecurityException](#) - if a security manager exists and its `checkExit` method doesn't allow the exit.

Since:

JDK1.1

See Also:

[exit\(int\)](#), [gc\(\)](#), [SecurityManager.checkExit\(int\)](#)

exec

```
public Process exec(String command)
    throws IOException
```

Executes the specified string command in a separate process.

This is a convenience method. An invocation of the form `exec(command)` behaves in exactly the same way as the invocation `exec(command, null, null)`.

Parameters:

command - a specified system command.

Returns:

A new [Process](#) object for managing the subprocess

Throws:

[SecurityException](#) - If a security manager exists and its [checkExec](#) method doesn't allow creation of the subprocess

[IOException](#) - If an I/O error occurs

[NullPointerException](#) - If command is null

[IllegalArgumentException](#) - If command is empty

See Also:

[exec\(String\[\], String\[\], File\)](#), [ProcessBuilder](#)

exec

```
public Process exec(String command,
    String[] envp)
    throws IOException
```

Executes the specified string command in a separate process with the specified environment.

This is a convenience method. An invocation of the form `exec(command, envp)` behaves in exactly the same way as the invocation `exec(command, envp, null)`.

Parameters:

command - a specified system command.

envp - array of strings, each element of which has environment variable settings in the format *name=value*, or null if the subprocess should inherit the environment of the

current process.

Returns:

A new [Process](#) object for managing the subprocess

Throws:

[SecurityException](#) - If a security manager exists and its [checkExec](#) method doesn't allow creation of the subprocess

[IOException](#) - If an I/O error occurs

[NullPointerException](#) - If `command` is `null`, or one of the elements of `envp` is `null`

[IllegalArgumentException](#) - If `command` is empty

See Also:

[exec\(String\[\], String\[\], File\)](#), [ProcessBuilder](#)

exec

```
public Process exec(String command,
                     String[] envp,
                     File dir)
    throws IOException
```

Executes the specified string command in a separate process with the specified environment and working directory.

This is a convenience method. An invocation of the form `exec(command, envp, dir)` behaves in exactly the same way as the invocation `exec(cmdarray, envp, dir)`, where `cmdarray` is an array of all the tokens in `command`.

More precisely, the `command` string is broken into tokens using a [StringTokenizer](#) created by the call `new StringTokenizer(command)` with no further modification of the character categories. The tokens produced by the tokenizer are then placed in the new string array `cmdarray`, in the same order.

Parameters:

`command` - a specified system command.

`envp` - array of strings, each element of which has environment variable settings in the format *name=value*, or `null` if the subprocess should inherit the environment of the current process.

`dir` - the working directory of the subprocess, or `null` if the subprocess should inherit the working directory of the current process.

Returns:

A new [Process](#) object for managing the subprocess

Throws:

[SecurityException](#) - If a security manager exists and its [checkExec](#) method doesn't allow creation of the subprocess

[IOException](#) - If an I/O error occurs

[NullPointerException](#) - If `command` is `null`, or one of the elements of `envp` is `null`

[IllegalArgumentException](#) - If `command` is empty

Since:

1.3

See Also:

[ProcessBuilder](#)

exec

```
public Process exec(String[] cmdarray)
    throws IOException
```

Executes the specified command and arguments in a separate process.

This is a convenience method. An invocation of the form `exec(cmdarray)` behaves in exactly the same way as the invocation `exec(cmdarray, null, null)`.

Parameters:

`cmdarray` - array containing the command to call and its arguments.

Returns:

A new [Process](#) object for managing the subprocess

Throws:

[SecurityException](#) - If a security manager exists and its [checkExec](#) method doesn't allow creation of the subprocess

[IOException](#) - If an I/O error occurs

[NullPointerException](#) - If `cmdarray` is `null`, or one of the elements of `cmdarray` is `null`

[IndexOutOfBoundsException](#) - If `cmdarray` is an empty array (has length 0)

See Also:

[ProcessBuilder](#)

exec

```
public Process exec(String[] cmdarray,
```

[String\[\]](#) envp)
throws [IOException](#)

Executes the specified command and arguments in a separate process with the specified environment.

This is a convenience method. An invocation of the form `exec(cmdarray, envp)` behaves in exactly the same way as the invocation `exec(cmdarray, envp, null)`.

Parameters:

`cmdarray` - array containing the command to call and its arguments.

`envp` - array of strings, each element of which has environment variable settings in the format *name=value*, or `null` if the subprocess should inherit the environment of the current process.

Returns:

A new [Process](#) object for managing the subprocess

Throws:

[SecurityException](#) - If a security manager exists and its [checkExec](#) method doesn't allow creation of the subprocess

[IOException](#) - If an I/O error occurs

[NullPointerException](#) - If `cmdarray` is `null`, or one of the elements of `cmdarray` is `null`, or one of the elements of `envp` is `null`

[IndexOutOfBoundsException](#) - If `cmdarray` is an empty array (has length 0)

See Also:

[ProcessBuilder](#)

exec

```
public Process exec(String[] cmdarray,  
                    String[] envp,  
                    File dir)  
    throws IOException
```

Executes the specified command and arguments in a separate process with the specified environment and working directory.

Given an array of strings `cmdarray`, representing the tokens of a command line, and an array of strings `envp`, representing "environment" variable settings, this method creates a new process in which to execute the specified command.

This method checks that `cmdarray` is a valid operating system command. Which commands

are valid is system-dependent, but at the very least the command must be a non-empty list of non-null strings.

If `envp` is `null`, the subprocess inherits the environment settings of the current process.

[`ProcessBuilder.start\(\)`](#) is now the preferred way to start a process with a modified environment.

The working directory of the new subprocess is specified by `dir`. If `dir` is `null`, the subprocess inherits the current working directory of the current process.

If a security manager exists, its [`checkExec`](#) method is invoked with the first component of the array `cmdarray` as its argument. This may result in a [`SecurityException`](#) being thrown.

Starting an operating system process is highly system-dependent. Among the many things that can go wrong are:

- The operating system program file was not found.
- Access to the program file was denied.
- The working directory does not exist.

In such cases an exception will be thrown. The exact nature of the exception is system-dependent, but it will always be a subclass of [`IOException`](#).

Parameters:

`cmdarray` - array containing the command to call and its arguments.

`envp` - array of strings, each element of which has environment variable settings in the format *name=value*, or `null` if the subprocess should inherit the environment of the current process.

`dir` - the working directory of the subprocess, or `null` if the subprocess should inherit the working directory of the current process.

Returns:

A new [`Process`](#) object for managing the subprocess

Throws:

[`SecurityException`](#) - If a security manager exists and its [`checkExec`](#) method doesn't allow creation of the subprocess

[`IOException`](#) - If an I/O error occurs

[`NullPointerException`](#) - If `cmdarray` is `null`, or one of the elements of `cmdarray` is `null`, or one of the elements of `envp` is `null`

[`IndexOutOfBoundsException`](#) - If `cmdarray` is an empty array (has length 0)

Since:

1.3

See Also:

[ProcessBuilder](#)

availableProcessors

```
public int availableProcessors()
```

Returns the number of processors available to the Java virtual machine.

This value may change during a particular invocation of the virtual machine. Applications that are sensitive to the number of available processors should therefore occasionally poll this property and adjust their resource usage appropriately.

Returns:

the maximum number of processors available to the virtual machine; never smaller than one

Since:

1.4

freeMemory

```
public long freeMemory()
```

Returns the amount of free memory in the Java Virtual Machine. Calling the `gc` method may result in increasing the value returned by `freeMemory`.

Returns:

an approximation to the total amount of memory currently available for future allocated objects, measured in bytes.

totalMemory

```
public long totalMemory()
```

Returns the total amount of memory in the Java virtual machine. The value returned by this method may vary over time, depending on the host environment.

Note that the amount of memory required to hold an object of any given type may be implementation-dependent.

Returns:

the total amount of memory currently available for current and future objects, measured in bytes.

maxMemory

```
public long maxMemory()
```

Returns the maximum amount of memory that the Java virtual machine will attempt to use. If there is no inherent limit then the value [Long.MAX_VALUE](#) will be returned.

Returns:

the maximum amount of memory that the virtual machine will attempt to use, measured in bytes

Since:

1.4

gc

```
public void gc()
```

Runs the garbage collector. Calling this method suggests that the Java virtual machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the virtual machine has made its best effort to recycle all discarded objects.

The name `gc` stands for "garbage collector". The virtual machine performs this recycling process automatically as needed, in a separate thread, even if the `gc` method is not invoked explicitly.

The method [System.gc\(\)](#) is the conventional and convenient means of invoking this method.

runFinalization

```
public void runFinalization()
```

Runs the finalization methods of any objects pending finalization. Calling this method suggests that the Java virtual machine expend effort toward running the `finalize` methods of objects that have been found to be discarded but whose `finalize` methods have not yet been run. When control returns from the method call, the virtual machine has made a best effort to complete all outstanding finalizations.

The virtual machine performs the finalization process automatically as needed, in a separate thread, if the `runFinalization` method is not invoked explicitly.

The method [System.runFinalization\(\)](#) is the conventional and convenient means of invoking this method.

See Also:

[Object.finalize\(\)](#)

traceInstructions

```
public void traceInstructions(boolean on)
```

Enables/Disables tracing of instructions. If the `boolean` argument is `true`, this method suggests that the Java virtual machine emit debugging information for each instruction in the virtual machine as it is executed. The format of this information, and the file or other output stream to which it is emitted, depends on the host environment. The virtual machine may ignore this request if it does not support this feature. The destination of the trace output is system dependent.

If the `boolean` argument is `false`, this method causes the virtual machine to stop performing the detailed instruction trace it is performing.

Parameters:

`on` - `true` to enable instruction tracing; `false` to disable this feature.

traceMethodCalls

```
public void traceMethodCalls(boolean on)
```

Enables/Disables tracing of method calls. If the `boolean` argument is `true`, this method suggests that the Java virtual machine emit debugging information for each method in the virtual machine as it is called. The format of this information, and the file or other output stream to which it is emitted, depends on the host environment. The virtual machine may

ignore this request if it does not support this feature.

Calling this method with argument `false` suggests that the virtual machine cease emitting per-call debugging information.

Parameters:

`on` - `true` to enable instruction tracing; `false` to disable this feature.

load

```
public void load(String filename)
```

Loads the specified filename as a dynamic library. The filename argument must be a complete path name. From `java_g` it will automatically insert `"_g"` before the `".so"` (for example `Runtime.getRuntime().load("/home/avh/lib/libX11.so");`).

First, if there is a security manager, its `checkLink` method is called with the `filename` as its argument. This may result in a security exception.

This is similar to the method [loadLibrary\(String\)](#), but it accepts a general file name as an argument rather than just a library name, allowing any file of native code to be loaded.

The method [System.load\(String\)](#) is the conventional and convenient means of invoking this method.

Parameters:

`filename` - the file to load.

Throws:

[SecurityException](#) - if a security manager exists and its `checkLink` method doesn't allow loading of the specified dynamic library

[UnsatisfiedLinkError](#) - if the file does not exist.

[NullPointerException](#) - if `filename` is `null`

See Also:

[getRuntime\(\)](#), [SecurityException](#), [SecurityManager.checkLink\(java.lang.String\)](#)

loadLibrary

```
public void loadLibrary(String libname)
```

Loads the dynamic library with the specified library name. A file containing native code is loaded from the local file system from a place where library files are conventionally obtained. The details of this process are implementation-dependent. The mapping from a library name to a specific filename is done in a system-specific manner.

First, if there is a security manager, its `checkLink` method is called with the `libname` as its argument. This may result in a security exception.

The method [System.loadLibrary\(String\)](#) is the conventional and convenient means of invoking this method. If native methods are to be used in the implementation of a class, a standard strategy is to put the native code in a library file (call it `LibFile`) and then to put a static initializer:

```
static { System.loadLibrary("LibFile"); }
```

within the class declaration. When the class is loaded and initialized, the necessary native code implementation for the native methods will then be loaded as well.

If this method is called more than once with the same library name, the second and subsequent calls are ignored.

Parameters:

`libname` - the name of the library.

Throws:

[SecurityException](#) - if a security manager exists and its `checkLink` method doesn't allow loading of the specified dynamic library

[UnsatisfiedLinkError](#) - if the library does not exist.

[NullPointerException](#) - if `libname` is `null`

See Also:

[SecurityException](#), [SecurityManager.checkLink\(java.lang.String\)](#)

getLocalizedInputStream

```
public InputStream getLocalizedInputStream(InputStream in)
```

Deprecated. *As of JDK 1.1, the preferred way to translate a byte stream in the local encoding into a character stream in Unicode is via the `InputStreamReader` and `BufferedReader` classes.*

Creates a localized version of an input stream. This method takes an `InputStream` and returns an `InputStream` equivalent to the argument in all respects except that it is localized: as characters in the local character set are read from the stream, they are automatically converted from the local character set to Unicode.

If the argument is already a localized stream, it may be returned as the result.

Parameters:

`in` - `InputStream` to localize

Returns:

a localized input stream

See Also:

[InputStream](#), [BufferedReader.BufferedReader\(java.io.Reader\)](#),
[InputStreamReader.InputStreamReader\(java.io.InputStream\)](#)

getLocalizedOutputStream

```
public OutputStream getLocalizedOutputStream(OutputStream out)
```

Deprecated. *As of JDK 1.1, the preferred way to translate a Unicode character stream into a byte stream in the local encoding is via the `OutputStreamWriter`, `BufferedWriter`, and `PrintWriter` classes.*

Creates a localized version of an output stream. This method takes an `OutputStream` and returns an `OutputStream` equivalent to the argument in all respects except that it is localized: as Unicode characters are written to the stream, they are automatically converted to the local character set.

If the argument is already a localized stream, it may be returned as the result.

Parameters:

`out` - `OutputStream` to localize

Returns:

a localized output stream

See Also:

[OutputStream](#), [BufferedWriter.BufferedReader\(java.io.Writer\)](#),
[OutputStreamWriter.OutputStreamWriter\(java.io.](#)
[OutputStream\)](#), [PrintWriter.PrintWriter\(java.io.OutputStream\)](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | FIELD | CONSTR | [METHOD](#) DETAIL: FIELD | CONSTR | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

java.lang

Class ProcessBuilder

[java.lang.Object](#)└─ [java.lang.ProcessBuilder](#)

```
public final class ProcessBuilder
```

extends [Object](#)

This class is used to create operating system processes.

Each `ProcessBuilder` instance manages a collection of process attributes. The [start\(\)](#) method creates a new [Process](#) instance with those attributes. The [start\(\)](#) method can be invoked repeatedly from the same instance to create new subprocesses with identical or related attributes.

Each process builder manages these process attributes:

- a *command*, a list of strings which signifies the external program file to be invoked and its arguments, if any. Which string lists represent a valid operating system command is system-dependent. For example, it is common for each conceptual argument to be an element in this list, but there are operating systems where programs are expected to tokenize command line strings themselves - on such a system a Java implementation might require commands to contain exactly two elements.
- an *environment*, which is a system-dependent mapping from *variables* to *values*. The initial value is a copy of the environment of the current process (see [System.getenv\(\)](#)).
- a *working directory*. The default value is the current working directory of the current process, usually the directory named by the system property `user.dir`.
- a *redirectErrorStream* property. Initially, this property is `false`, meaning that the standard output and error output of a subprocess are sent to two separate streams, which can be accessed using the [Process.getInputStream\(\)](#) and [Process.getErrorStream\(\)](#) methods. If the value is set to `true`, the standard error is merged with the standard output. This makes it easier to correlate error messages with the corresponding output. In this case, the merged data can be read from the stream returned by [Process.getInputStream\(\)](#),

while reading from the stream returned by [Process.getErrorStream\(\)](#) will get an immediate end of file.

Modifying a process builder's attributes will affect processes subsequently started by that object's [start\(\)](#) method, but will never affect previously started processes or the Java process itself.

Most error checking is performed by the [start\(\)](#) method. It is possible to modify the state of an object so that [start\(\)](#) will fail. For example, setting the command attribute to an empty list will not throw an exception unless [start\(\)](#) is invoked.

Note that this class is not synchronized. If multiple threads access a `ProcessBuilder` instance concurrently, and at least one of the threads modifies one of the attributes structurally, it *must* be synchronized externally.

Starting a new process which uses the default working directory and environment is easy:

```
Process p = new ProcessBuilder("myCommand", "myArg").
start();
```

Here is an example that starts a process with a modified working directory and environment:

```
ProcessBuilder pb = new ProcessBuilder("myCommand",
"myArg1", "myArg2");
Map<String, String> env = pb.environment();
env.put("VAR1", "myValue");
env.remove("OTHERVAR");
env.put("VAR2", env.get("VAR1") + "suffix");
pb.directory("myDir");
Process p = pb.start();
```

To start a process with an explicit set of environment variables, first call [Map.clear\(\)](#) before adding environment variables.

Since:

1.5

Constructor Summary

[ProcessBuilder](#)([List<String>](#) command)

Constructs a process builder with the specified operating system program and arguments.

[ProcessBuilder](#)([String](#)... command)

Constructs a process builder with the specified operating system program and arguments.

Method Summary

List < String >	command () Returns this process builder's operating system program and arguments.
ProcessBuilder	command (List < String > command) Sets this process builder's operating system program and arguments.
ProcessBuilder	command (String ... command) Sets this process builder's operating system program and arguments.
File	directory () Returns this process builder's working directory.
ProcessBuilder	directory (File directory) Sets this process builder's working directory.
Map < String , String >	environment () Returns a string map view of this process builder's environment.
boolean	redirectErrorStream () Tells whether this process builder merges standard error and standard output.
ProcessBuilder	redirectErrorStream (boolean redirectErrorStream) Sets this process builder's <code>redirectErrorStream</code> property.
Process	start () Starts a new process using the attributes of this process builder.

Methods inherited from class [java.lang.Object](#)

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#),
[toString](#), [wait](#), [wait](#), [wait](#)

Constructor Detail**ProcessBuilder**

```
public ProcessBuilder(List<String> command)
```

Constructs a process builder with the specified operating system program and arguments. This

constructor does *not* make a copy of the `command` list. Subsequent updates to the list will be reflected in the state of the process builder. It is not checked whether `command` corresponds to a valid operating system command.

Parameters:

`command` - The list containing the program and its arguments

Throws:

[NullPointerException](#) - If the argument is `null`

ProcessBuilder

```
public ProcessBuilder(String... command)
```

Constructs a process builder with the specified operating system program and arguments. This is a convenience constructor that sets the process builder's `command` to a string list containing the same strings as the `command` array, in the same order. It is not checked whether `command` corresponds to a valid operating system command.

Parameters:

`command` - A string array containing the program and its arguments

Method Detail

`command`

```
public ProcessBuilder command(List<String> command)
```

Sets this process builder's operating system program and arguments. This method does *not* make a copy of the `command` list. Subsequent updates to the list will be reflected in the state of the process builder. It is not checked whether `command` corresponds to a valid operating system command.

Parameters:

`command` - The list containing the program and its arguments

Returns:

This process builder

Throws:

[NullPointerException](#) - If the argument is `null`

command

```
public ProcessBuilder command(String... command)
```

Sets this process builder's operating system program and arguments. This is a convenience method that sets the command to a string list containing the same strings as the `command` array, in the same order. It is not checked whether `command` corresponds to a valid operating system command.

Parameters:

`command` - A string array containing the program and its arguments

Returns:

This process builder

command

```
public List<String> command()
```

Returns this process builder's operating system program and arguments. The returned list is *not* a copy. Subsequent updates to the list will be reflected in the state of this process builder.

Returns:

This process builder's program and its arguments

environment

```
public Map<String, String> environment()
```

Returns a string map view of this process builder's environment. Whenever a process builder is created, the environment is initialized to a copy of the current process environment (see [System.getenv\(\)](#)). Subprocesses subsequently started by this object's [start\(\)](#) method will use this map as their environment.

The returned object may be modified using ordinary [Map](#) operations. These modifications will be visible to subprocesses started via the [start\(\)](#) method. Two `ProcessBuilder` instances always contain independent process environments, so changes to the returned map will never be reflected in any other `ProcessBuilder` instance or the values returned by [System.getenv](#).

If the system does not support environment variables, an empty map is returned.

The returned map does not permit null keys or values. Attempting to insert or query the presence of a null key or value will throw a [NullPointerException](#). Attempting to query the presence of a key or value which is not of type [String](#) will throw a [ClassCastException](#).

The behavior of the returned map is system-dependent. A system may not allow modifications to environment variables or may forbid certain variable names or values. For this reason, attempts to modify the map may fail with [UnsupportedOperationException](#) or [IllegalArgumentException](#) if the modification is not permitted by the operating system.

Since the external format of environment variable names and values is system-dependent, there may not be a one-to-one mapping between them and Java's Unicode strings. Nevertheless, the map is implemented in such a way that environment variables which are not modified by Java code will have an unmodified native representation in the subprocess.

The returned map and its collection views may not obey the general contract of the [Object.equals\(java.lang.Object\)](#) and [Object.hashCode\(\)](#) methods.

The returned map is typically case-sensitive on all platforms.

If a security manager exists, its [checkPermission](#) method is called with a [RuntimePermission\("getenv.*"\)](#) permission. This may result in a [SecurityException](#) being thrown.

When passing information to a Java subprocess, [system properties](#) are generally preferred over environment variables.

Returns:

This process builder's environment

Throws:

[SecurityException](#) - If a security manager exists and its [checkPermission](#) method doesn't allow access to the process environment

See Also:

[Runtime.exec\(String\[\],String\[\],java.io.File\)](#), [System.getenv\(\)](#)

```
public File directory()
```

Returns this process builder's working directory. Subprocesses subsequently started by this object's [start\(\)](#) method will use this as their working directory. The returned value may be `null` -- this means to use the working directory of the current Java process, usually the directory named by the system property `user.dir`, as the working directory of the child process.

Returns:

This process builder's working directory

directory

```
public ProcessBuilder directory(File directory)
```

Sets this process builder's working directory. Subprocesses subsequently started by this object's [start\(\)](#) method will use this as their working directory. The argument may be `null` -- this means to use the working directory of the current Java process, usually the directory named by the system property `user.dir`, as the working directory of the child process.

Parameters:

`directory` - The new working directory

Returns:

This process builder

redirectErrorStream

```
public boolean redirectErrorStream()
```

Tells whether this process builder merges standard error and standard output.

If this property is `true`, then any error output generated by subprocesses subsequently started by this object's [start\(\)](#) method will be merged with the standard output, so that both can be read using the [Process.getInputStream\(\)](#) method. This makes it easier to correlate error messages with the corresponding output. The initial value is `false`.

Returns:

This process builder's `redirectErrorStream` property

redirectErrorStream

```
public ProcessBuilder redirectErrorStream  
(boolean redirectErrorStream)
```

Sets this process builder's `redirectErrorStream` property.

If this property is `true`, then any error output generated by subprocesses subsequently started by this object's [start\(\)](#) method will be merged with the standard output, so that both can be read using the [Process.getInputStream\(\)](#) method. This makes it easier to correlate error messages with the corresponding output. The initial value is `false`.

Parameters:

`redirectErrorStream` - The new property value

Returns:

This process builder

start

```
public Process start()  
    throws IOException
```

Starts a new process using the attributes of this process builder.

The new process will invoke the command and arguments given by [command\(\)](#), in a working directory as given by [directory\(\)](#), with a process environment as given by [environment\(\)](#).

This method checks that the command is a valid operating system command. Which commands are valid is system-dependent, but at the very least the command must be a non-empty list of non-null strings.

If there is a security manager, its [checkExec](#) method is called with the first component of this object's `command` array as its argument. This may result in a [SecurityException](#) being thrown.

Starting an operating system process is highly system-dependent. Among the many things that can go wrong are:

- The operating system program file was not found.
- Access to the program file was denied.
- The working directory does not exist.

In such cases an exception will be thrown. The exact nature of the exception is system-dependent, but it will always be a subclass of [IOException](#).

Subsequent modifications to this process builder will not affect the returned [Process](#).

Returns:

A new [Process](#) object for managing the subprocess

Throws:

[NullPointerException](#) - If an element of the command list is null

[IndexOutOfBoundsException](#) - If the command is an empty list (has size 0)

[SecurityException](#) - If a security manager exists and its [checkExec](#) method doesn't allow creation of the subprocess

[IOException](#) - If an I/O error occurs

See Also:

[Runtime.exec\(String\[\], String\[\], java.io.File\)](#),
[SecurityManager.checkExec\(String\)](#)

[Overview](#) **[Package](#)** **[Class](#)** **[Use](#)** **[Tree](#)** **[Deprecated](#)** **[Index](#)** **[Help](#)**

*JavaTM 2 Platform
Std. Ed. v1.5.0*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).


```
import java.io.BufferedReader;
import java.io.IOException;
public class JavaExec {
    public static void main(String args[]) {
        Process p = null;
        System.out.println("creating child");
        Runtime rt = Runtime.getRuntime();
        String command[] = new String[2];
        command[0] = "java";
        command[1] = "Child";
        try {
            p = rt.exec(command);
        } catch (IOException ioe) {
            System.out.println("IOException occurred");
        }
        try {
            p.waitFor();
        } catch (InterruptedException ie) {
            System.out.println("InterruptedException occurred");
        }
        System.out.println("child has finished execution");
        BufferedReader pOut = new BufferedReader(p.getInputStream());
        byte output[] = null;
        try {
            output = new byte[pOut.available()];
            pOut.read(output);
        } catch (IOException ioe) {
            System.out.println("IOException occurred");
        }
        System.out.println("output of the child process was: "
            + new String(output));
    }
}
```

creating child

child has finished execution

output of the child process was: hello from the child!

java.lang

Class Process

[java.lang.Object](#)└─ [java.lang.Process](#)

```
public abstract class Process
```

extends [Object](#)

The [ProcessBuilder.start\(\)](#) and [Runtime.exec](#) methods create a native process and return an instance of a subclass of **Process** that can be used to control the process and obtain information about it. The class **Process** provides methods for performing input from the process, performing output to the process, waiting for the process to complete, checking the exit status of the process, and destroying (killing) the process.

The methods that create processes may not work well for special processes on certain native platforms, such as native windowing processes, daemon processes, Win16/DOS processes on Microsoft Windows, or shell scripts. The created subprocess does not have its own terminal or console. All its standard io (i.e. stdin, stdout, stderr) operations will be redirected to the parent process through three streams ([getOutputStream\(\)](#), [getInputStream\(\)](#), [getErrorStream\(\)](#)). The parent process uses these streams to feed input to and get output from the subprocess. Because some native platforms only provide limited buffer size for standard input and output streams, failure to promptly write the input stream or read the output stream of the subprocess may cause the subprocess to block, and even deadlock.

The subprocess is not killed when there are no more references to the **Process** object, but rather the subprocess continues executing asynchronously.

There is no requirement that a process represented by a **Process** object execute asynchronously or concurrently with respect to the Java process that owns the **Process** object.

Since:

JDK1.0

See Also:[ProcessBuilder](#), [Runtime.exec\(String\[\], String\[\], File\)](#)

Constructor Summary

[Process](#)()

Method Summary

abstract void	<u>destroy</u> () Kills the subprocess.
abstract int	<u>exitValue</u> () Returns the exit value for the subprocess.
abstract <u>InputStream</u>	<u>getErrorStream</u> () Gets the error stream of the subprocess.
abstract <u>InputStream</u>	<u>getInputStream</u> () Gets the input stream of the subprocess.
abstract <u>OutputStream</u>	<u>getOutputStream</u> () Gets the output stream of the subprocess.
abstract int	<u>waitFor</u> () causes the current thread to wait, if necessary, until the process represented by this Process object has terminated.

Methods inherited from class java.lang.[Object](#)

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

Constructor Detail

Process

```
public Process( )
```

Method Detail

getOutputStream

```
public abstract OutputStream getOutputStream()
```

Gets the output stream of the subprocess. Output to the stream is piped into the standard input stream of the process represented by this `Process` object.

Implementation note: It is a good idea for the output stream to be buffered.

Returns:

the output stream connected to the normal input of the subprocess.

getInputStream

```
public abstract InputStream getInputStream()
```

Gets the input stream of the subprocess. The stream obtains data piped from the standard output stream of the process represented by this `Process` object.

Implementation note: It is a good idea for the input stream to be buffered.

Returns:

the input stream connected to the normal output of the subprocess.

See Also:

[ProcessBuilder.redirectErrorStream\(\)](#)

getErrorStream

```
public abstract InputStream getErrorStream()
```

Gets the error stream of the subprocess. The stream obtains data piped from the error output stream of the process represented by this `Process` object.

Implementation note: It is a good idea for the input stream to be buffered.

Returns:

the input stream connected to the error stream of the subprocess.

See Also:

[ProcessBuilder.redirectErrorStream\(\)](#)

waitFor

```
public abstract int waitFor()  
    throws InterruptedException
```

causes the current thread to wait, if necessary, until the process represented by this `Process` object has terminated. This method returns immediately if the subprocess has already terminated. If the subprocess has not yet terminated, the calling thread will be blocked until the subprocess exits.

Returns:

the exit value of the process. By convention, 0 indicates normal termination.

Throws:

[InterruptedException](#) - if the current thread is [interrupted](#) by another thread while it is waiting, then the wait is ended and an [InterruptedException](#) is thrown.

exitValue

```
public abstract int exitValue()
```

Returns the exit value for the subprocess.

Returns:

the exit value of the subprocess represented by this `Process` object. by convention, the value 0 indicates normal termination.

Throws:

[IllegalThreadStateException](#) - if the subprocess represented by this `Process` object has not yet terminated.

destroy

```
public abstract void destroy()
```

Kills the subprocess. The subprocess represented by this `Process` object is forcibly terminated.

[Overview](#) **[Package](#)** **[Class](#)** **[Use Tree](#)** **[Deprecated](#)** **[Index](#)** **[Help](#)***JavaTM 2 Platform
Std. Ed. v1.5.0***[PREV CLASS](#)** **[NEXT CLASS](#)****[FRAMES](#)****[NO FRAMES](#)****[All Classes](#)**SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

```
#include <stdio.h>
#include <pthread.h>

int counter=0;

void threadedFunction(void *ptr){
    int i;
    for (i=0; i<10; i++) {
        printf("hello from %s (counter=%d)\n",(char*) ptr,counter++);
        sleep(1);
    }
}

int main(int argc, char** argv) {
    pthread_t thread1, thread2;
    int  iret1, iret2;

    iret1 = pthread_create( &thread1, NULL, (void*)&threadedFunction, (void*) "thread 1");
    iret2 = pthread_create( &thread2, NULL, (void*)&threadedFunction, (void*) "thread 2");

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("thread 1 returns: %d\n",iret1);
    printf("thread 2 returns: %d\n",iret2);
    exit(0);
}
```



```
hello from thread 1 (counter=0)
hello from thread 2 (counter=1)
hello from thread 1 (counter=2)
hello from thread 2 (counter=3)
hello from thread 1 (counter=4)
hello from thread 2 (counter=5)
hello from thread 1 (counter=6)
hello from thread 2 (counter=7)
hello from thread 1 (counter=8)
hello from thread 2 (counter=9)
hello from thread 1 (counter=10)
hello from thread 2 (counter=11)
hello from thread 1 (counter=12)
hello from thread 2 (counter=13)
hello from thread 1 (counter=14)
hello from thread 2 (counter=15)
hello from thread 1 (counter=16)
hello from thread 2 (counter=17)
hello from thread 1 (counter=18)
hello from thread 2 (counter=19)
thread 1 returns: 0
thread 2 returns: 0
```

```
int main(int argc, char** argv) {
    int i,j, pid, result;

    for (j=0; j<10;j++) {
        if ((pid=vfork()) == -1) {
            printf("error creating process\n");
            exit(1);
        }
        //waitpid(pid, &result,0);
    }

    return 0;
}
```

```
#include <stdio.h>
#include <pthread.h>

int a,b,c;

void threadedFunction(){
    int i;
    for (i=0; i<1000; i++)
        printf("!");
}

int main(int argc, char** argv) {
    pthread_t thread1;
    int  iret1;
    int cnt;

    for (cnt=0; cnt<10;cnt++) {
        iret1 = pthread_create( &thread1, NULL, (void*)&threadedFunction, NULL);
        //pthread_join( thread1, NULL);
    }

    exit(0);
}
```



Search for

MSDN Library

Go

Advanced Search

sync toc ↻ x

- Up One Level
- AssignProcessToJobObject
- AttachThreadInput
- BindIoCompletionCallback
- CommandLineToArgvW
- ConvertFiberToThread
- ConvertThreadToFiber
- ConvertThreadToFiberEx
- CreateFiber
- CreateFiberEx
- CreateJobObject
- CreateProcess
- CreateProcessAsUser
- CreateProcessWithLogonW
- CreateProcessWithTokenW
- CreateRemoteThread
- CreateThread
- DeleteFiber
- ExitProcess
- ExitThread
- FiberProc
- FlsAlloc
- FlsCallback
- FlsFree
- FlsGetValue
- FlsSetValue
- FreeEnvironmentStrings
- GetCommandLine
- GetCurrentProcess
- GetCurrentProcessId
- GetCurrentThread
- GetCurrentThreadId
- GetEnvironmentStrings
- GetEnvironmentVariable

Welcome to the MSDN Library

[MSDN Home](#) > [MSDN Library](#) > [Windows Development](#) > [Windows Base Services](#) > [DLLs, Processes, and Threads](#) > [DLLs, Processes, and Threads](#) > [Processes and Threads](#) > [Process and Thread Reference](#) > [Process and Thread Functions](#)

Platform SDK: DLLs, Processes, and Threads

CreateThread

The **CreateThread** function creates a thread to execute within the virtual address space of the calling process.

To create a thread that runs in the virtual address space of another process, use the [CreateRemoteThread](#) function.

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);
```

Parameters

lpThreadAttributes
[in] Pointer to a [SECURITY_ATTRIBUTES](#) structure that determines whether the returned handle can be inherited by child processes. If *lpThreadAttributes* is NULL, the handle cannot be inherited.

The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new thread. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor. The ACLs in the default security descriptor for a thread come from the primary or impersonation token of the creator.

dwStackSize
[in] Initial size of the stack, in bytes. The system rounds this value to the nearest page. If this parameter is zero, the new thread uses the default size for the executable. For more information, see [Thread Stack Size](#).

lpStartAddress
[in] Pointer to the application-defined function to be executed by the thread and represents the starting address of the thread. For more information on the thread function, see [ThreadProc](#).

lpParameter
[in] Pointer to a variable to be passed to the thread.

dwCreationFlags
[in] Flags that control the creation of the thread. If the CREATE_SUSPENDED flag is specified, the thread is created in a suspended state, and will not run until the [ResumeThread](#) function is called. If this value is zero, the thread runs immediately after creation.

If the STACK_SIZE_PARAM_IS_A_RESERVATION flag is specified, the *dwStackSize* parameter specifies the initial reserve size of the stack. Otherwise, *dwStackSize* specifies the commit size.

Windows 2000/NT and Windows Me/98/95: The STACK_SIZE_PARAM_IS_A_RESERVATION flag is not supported.

lpThreadId
[out] Pointer to a variable that receives the thread identifier. If this parameter is NULL, the thread identifier is not returned.

Windows Me/98/95: This parameter may not be NULL.

- ☐ [GetExitCodeProcess](#)
- ☐ [GetExitCodeThread](#)
- ☐ [GetGuiResources](#)
- ☐ [GetCurrentProcessorNumber](#)
- ☐ [GetLogicalProcessorInformation](#)
- ☐ [GetNumaAvailableMemoryNode](#)
- ☐ [GetNumaHighestNodeNumber](#)
- ☐ [GetNumaNodeProcessorMask](#)
- ☐ [GetNumaProcessorNode](#)
- ☐ [GetPriorityClass](#)
- ☐ [GetProcessAffinityMask](#)
- ☐ [GetProcessHandleCount](#)
- ☐ [GetProcessId](#)
- ☐ [GetProcessIdOfThread](#)
- ☐ [GetProcessIoCounters](#)
- ☐ [GetProcessPriorityBoost](#)
- ☐ [GetProcessShutdownParameters](#)
- ☐ [GetProcessTimes](#)
- ☐ [GetProcessVersion](#)
- ☐ [GetProcessWorkingSetSize](#)
- ☐ [GetProcessWorkingSetSizeEx](#)
- ☐ [GetStartupInfo](#)
- ☐ [GetThreadId](#)
- ☐ [GetThreadIoPendingFlag](#)
- ☐ [GetThreadPriority](#)
- ☐ [GetThreadPriorityBoost](#)
- ☐ [GetThreadStartInformation](#)
- ☐ [GetThreadTimes](#)
- ☐ [IsProcessInJob](#)
- ☐ [NeedCurrentDirectoryForExePath](#)
- ☐ [NtQueryInformationProcess](#)
- ☐ [NtQueryInformationThread](#)
- ☐ [OpenJobObject](#)
- ☐ [OpenProcess](#)
- ☐ [OpenThread](#)
- ☐ [QueryInformationJobObject](#)
- ☐ [QueueUserWorkItem](#)
- ☐ [ResumeThread](#)
- ☐ [SetEnvironmentVariable](#)
- ☐ [SetInformationJobObject](#)
- ☐ [SetPriorityClass](#)
- ☐ [SetProcessAffinityMask](#)
- ☐ [SetProcessPriorityBoost](#)
- ☐ [SetProcessShutdownParameters](#)
- ☐ [SetProcessWorkingSetSize](#)
- ☐ [SetProcessWorkingSetSizeEx](#)
- ☐ [SetThreadAffinityMask](#)
- ☐ [SetThreadIdealProcessor](#)

Return Values

If the function succeeds, the return value is a handle to the new thread.

If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

Note that **CreateThread** may succeed even if *lpStartAddress* points to data, code, or is not accessible. If the start address is invalid when the thread runs, an exception occurs, and the thread terminates. Thread termination due to a invalid start address is handled as an error exit for the thread's process. This behavior is similar to the asynchronous nature of [CreateProcess](#), where the process is created even if it refers to invalid or missing dynamic-link libraries (DLLs).

Windows Me/98/95: **CreateThread** succeeds only when it is called in the context of a 32-bit program. A 32-bit DLL cannot create an additional thread when that DLL is being called by a 16-bit program.

Remarks

The number of threads a process can create is limited by the available virtual memory. By default, every thread has one megabyte of stack space. Therefore, you can create at most 2028 threads. If you reduce the default stack size, you can create more threads. However, your application will have better performance if you create one thread per processor and build queues of requests for which the application maintains the context information. A thread would process all requests in a queue before processing requests in the next queue.

The new thread handle is created with the `THREAD_ALL_ACCESS` access right. If a security descriptor is not provided, the handle can be used in any function that requires a thread object handle. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If the access check denies access, the requesting process cannot use the handle to gain access to the thread. If the thread impersonates a client, then calls **CreateThread** with a NULL security descriptor, the thread object created has a default security descriptor which allows access only to the impersonation token's `TokenDefaultDacl` owner or members. For more information, see [Thread Security and Access Rights](#).

The thread execution begins at the function specified by the *lpStartAddress* parameter. If this function returns, the **DWORD** return value is used to terminate the thread in an implicit call to the [ExitThread](#) function. Use the [GetExitCodeThread](#) function to get the thread's return value.

The thread is created with a thread priority of `THREAD_PRIORITY_NORMAL`. Use the [GetThreadPriority](#) and [SetThreadPriority](#) functions to get and set the priority value of a thread.

When a thread terminates, the thread object attains a signaled state, satisfying any threads that were waiting on the object.

The thread object remains in the system until the thread has terminated and all handles to it have been closed through a call to [CloseHandle](#).

The [ExitProcess](#), [ExitThread](#), **CreateThread**, [CreateRemoteThread](#) functions, and a process that is starting (as the result of a call by **CreateProcess**) are serialized between each other within a process. Only one of these events can happen in an address space at a time. This means that the following restrictions hold:

Do not create a thread while impersonating another user. The call will succeed, however the newly created thread will have reduced access rights to itself when calling [GetCurrentThread](#). The access rights granted are derived from the access rights that the impersonated user has to the process. Some access rights including `THREAD_SET_THREAD_TOKEN` and `THREAD_GET_CONTEXT` may not be present, leading to unexpected failures.

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is done for the process.
- Only one thread in a process can be in a DLL initialization or detach routine at a time.
- **ExitProcess** does not return until no threads are in their DLL initialization or detach routines.

A thread in an executable that is linked to the static C run-time library (CRT) should use **_beginthread** and **_endthread** for thread management rather than **CreateThread** and

- ☐ [SetThreadPriority](#)
- ☐ [SetThreadPriorityBoost](#)
- ☐ [Sleep](#)
- ☐ [SleepEx](#)
- ☐ [SuspendThread](#)
- ☐ [SwitchToFiber](#)
- ☐ [SwitchToThread](#)
- ☐ [TerminateJobObject](#)
- ☐ [TerminateProcess](#)
- ☐ [TerminateThread](#)
- ☐ [ThreadProc](#)
- ☐ [TlsAlloc](#)
- ☐ [TlsFree](#)
- ☐ [TlsGetValue](#)
- ☐ [TlsSetValue](#)
- ☐ [UserHandleGrantAccess](#)
- ☐ [WaitForInputIdle](#)
- ☐ [WinExec](#)

ExitThread. Failure to do so results in small memory leaks when the thread calls **ExitThread**. Another work around is to link the executable to the CRT in a DLL instead of the static CRT. Note that this memory leak only occurs from a DLL if the DLL is linked to the static CRT and a thread calls the [DisableThreadLibraryCalls](#) function. Otherwise, it is safe to call **CreateThread** and **ExitThread** from a thread in a DLL that links to the static CRT.

Example Code

For an example, see [Creating Threads](#).

Requirements

Client: Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.
Server: Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

[Processes and Threads Overview](#), [Process and Thread Functions](#), [CloseHandle](#), [CreateProcess](#), [CreateRemoteThread](#), [ExitProcess](#), [ExitThread](#), [GetExitCodeThread](#), [GetThreadPriority](#), [ResumeThread](#), [SetThreadPriority](#), [SECURITY_ATTRIBUTES](#), [ThreadProc](#)

Last updated: May 2004 | [What did you think of this topic?](#) | [Order a Platform SDK CD](#)
[© 2004 Microsoft Corporation. All rights reserved. Terms of use.](#)

[Manage Your Profile](#) | [Legal](#) | [Contact Us](#) | [MSDN Flash Newsletter](#)

©2004 Microsoft Corporation. All rights reserved. [Terms of Use](#) | [Trademarks](#) | [Privacy Statement](#)



```
#include <windows.h>

int counter=0;

DWORD WINAPI ThreadFunc(LPVOID lpParam) {
    int i;

    for (i=0;i<10;i++)
        printf("hello from thread %d (counter=%d)\n",*(DWORD*)lpParam, counter++);
    return 0;
}

int main(VOID) {
    DWORD dwThreadId1, dwThreadId2;
    DWORD dwThrdParam1=1, dwThrdParam2=2;
    HANDLE hThread1, hThread2;

    hThread1 = CreateThread(
        NULL,                // default security attributes
        0,                   // use default stack size
        ThreadFunc,          // thread function
        &dwThrdParam1,        // argument to thread function
        0,                   // use default creation flags
        &dwThreadId1);       // returns the thread identifier

    hThread2 = CreateThread(NULL,0,ThreadFunc,&dwThrdParam2,0,&dwThreadId2);

    if (hThread1 == NULL || hThread2 == NULL) {
        printf("CreateThread failed");
    } else {
        CloseHandle(hThread1);
        CloseHandle(hThread2);
    }

    Sleep(100);
    exit(0);
}
```

hello from thread 1 (counter=0)
hello from thread 1 (counter=1)
hello from thread 1 (counter=2)
hello from thread 2 (counter=3)
hello from thread 2 (counter=4)
hello from thread 2 (counter=5)
hello from thread 2 (counter=6)
hello from thread 2 (counter=7)
hello from thread 2 (counter=8)
hello from thread 1 (counter=9)
hello from thread 1 (counter=10)
hello from thread 1 (counter=11)
hello from thread 2 (counter=12)
hello from thread 2 (counter=13)
hello from thread 2 (counter=14)
hello from thread 2 (counter=15)
hello from thread 1 (counter=16)
hello from thread 1 (counter=17)
hello from thread 1 (counter=18)
hello from thread 1 (counter=19)



Search for

MSDN Library

Go

[Advanced Search](#)

sync toc

- Up One Level
- BindHandle Method
- GetAvailableThreads Method
- GetMaxThreads Method
- GetMinThreads Method
- QueueUserWorkItem Method
- RegisterWaitForSingleObject Method
- SetMinThreads Method
- UnsafeQueueUserWorkItem Method
- UnsafeRegisterWaitForSingleObject Method

Welcome to the MSDN Library

[MSDN Home](#) > [MSDN Library](#) > [.NET Development](#) > [.NET Framework SDK](#) > [.NET Framework](#) > [Reference](#) > [Class Library](#) > [System.Threading](#) > [ThreadPool Class](#)

.NET Framework Class Library

ThreadPool Methods

The methods of the **ThreadPool** class are listed here. For a complete list of **ThreadPool** class members, see the [ThreadPool Members](#) topic.

Public Methods

BindHandle	Binds an operating system handle to the ThreadPool .
GetAvailableThreads	Retrieves the difference between the maximum number of thread pool threads, returned by GetMaxThreads , and the number currently active.
GetMaxThreads	Retrieves the number of requests to the thread pool that can be active concurrently. All requests above that number remain queued until thread pool threads become available.
GetMinThreads	Retrieves the number of idle threads the ThreadPool maintains in anticipation of new requests.
QueueUserWorkItem Supported by the .NET Compact Framework.	Overloaded. Queues a method for execution. The method executes when a thread pool thread becomes available.
RegisterWaitForSingleObject	Overloaded. Registers a delegate that is waiting for a WaitHandle .
SetMinThreads	Sets the number of idle threads the ThreadPool maintains in anticipation of new requests.
UnsafeQueueUserWorkItem	Queues a user work item to the thread pool.
UnsafeRegisterWaitForSingleObject	Overloaded. Queues the specified delegate to the thread pool.

See Also

[ThreadPool Class](#) | [System.Threading Namespace](#) | [Managed Extensions for C++ Programming](#)

*Syntax based on .NET Framework version 1.1.
Documentation version 1.1.1.*

[Send comments on this topic.](#)

© Microsoft Corporation. All rights reserved.

[Manage Your Profile](#) | [Legal](#) | [Contact Us](#) | [MSDN Flash Newsletter](#)

©2004 Microsoft Corporation. All rights reserved. [Terms of Use](#) | [Trademarks](#) | [Privacy Statement](#)



```
using System;
using System.Threading;

class netThread {
    private static int counter=0;
    [STAThread]
    static void Main(string[] args) {
        ThreadPool.QueueUserWorkItem(new WaitCallback(myThread),"thread 1");
        ThreadPool.QueueUserWorkItem(new WaitCallback(myThread),"thread 2");
        Thread.Sleep(15000);
    }

    static void myThread(object threadName) {
        for (int i=0;i<10;i++) {
            Console.WriteLine("hello from "+threadName+ " (counter="+counter.ToString()+")");
            counter++;
            Thread.Sleep(1000);
        }
    }
}
```

hello from thread 1 (counter=0)
hello from thread 1 (counter=1)
hello from thread 2 (counter=2)
hello from thread 1 (counter=3)
hello from thread 2 (counter=4)
hello from thread 1 (counter=5)
hello from thread 2 (counter=6)
hello from thread 1 (counter=7)
hello from thread 2 (counter=8)
hello from thread 1 (counter=9)
hello from thread 2 (counter=10)
hello from thread 1 (counter=11)
hello from thread 2 (counter=12)
hello from thread 1 (counter=13)
hello from thread 2 (counter=14)
hello from thread 1 (counter=15)
hello from thread 2 (counter=16)
hello from thread 1 (counter=17)
hello from thread 2 (counter=18)
hello from thread 2 (counter=19)

java.lang

Interface Runnable

All Known Implementing Classes:

[AsyncBoxView.ChildState](#), [FutureTask](#), [RenderableImageProducer](#), [Thread](#), [TimerTask](#)

```
public interface Runnable
```

The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, `Runnable` is implemented by class `Thread`. Being active simply means that a thread has been started and has not yet been stopped.

In addition, `Runnable` provides the means for a class to be active while not subclassing `Thread`. A class that implements `Runnable` can run without subclassing `Thread` by instantiating a `Thread` instance and passing itself in as the target. In most cases, the `Runnable` interface should be used if you are only planning to override the `run()` method and no other `Thread` methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

Since:

JDK1.0

See Also:

[Thread](#)

Method Summary

void	run()

	When an object implementing interface <code>Runnable</code> is used to create a thread, starting the thread causes the object's <code>run</code> method to be called in that separately executing thread.

Method Detail

run

```
void run()
```

When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

The general contract of the method `run` is that it may take any action whatsoever.

See Also:

[Thread.run\(\)](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.5.0*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | FIELD | CONSTR | [METHOD](#) DETAIL: FIELD | CONSTR | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

```
public class JavaThreadExample {
    public static int counter = 0;
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyThread("thread 1"));
        Thread t2 = new Thread(new MyThread("thread 2"));
        t1.start();
        t2.start();
        System.out.println("threads started ...");
    }
}

class MyThread implements Runnable {
    protected String text;
    public MyThread(String text) {
        this.text = text;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("hello from " + text + " (counter="
                               + JavaThreadExample.counter + ")");
            JavaThreadExample.counter++;
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ie) {
                System.out.println("An InterruptedException occurred\n"
                                   + ie.toString() + "\n" + ie.getMessage());
            }
        }
    }
}
```

threads started ...

hello from thread 1 (counter=0)
hello from thread 2 (counter=1)
hello from thread 1 (counter=2)
hello from thread 2 (counter=3)
hello from thread 1 (counter=4)
hello from thread 2 (counter=5)
hello from thread 1 (counter=6)
hello from thread 2 (counter=7)
hello from thread 1 (counter=8)
hello from thread 2 (counter=9)
hello from thread 1 (counter=10)
hello from thread 2 (counter=10)
hello from thread 1 (counter=12)
hello from thread 2 (counter=13)
hello from thread 1 (counter=14)
hello from thread 2 (counter=15)
hello from thread 1 (counter=16)
hello from thread 2 (counter=17)
hello from thread 1 (counter=18)
hello from thread 2 (counter=19)

java.lang

Class Thread

[java.lang.Object](#)└─ [java.lang.Thread](#)

All Implemented Interfaces:

[Runnable](#)

```
public class Thread
```

extends [Object](#)implements [Runnable](#)

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new `Thread` object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named `main` of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The `exit` method of class `Runtime` has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the `run` method or by throwing an exception that propagates beyond the `run` method.

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of `Thread`. This subclass should override the `run` method of class `Thread`. An instance of the subclass can then be allocated and started. For example, a thread that computes primes larger than a stated value could be written as follows:


```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
        . . .
    }
}
```

The following code would then create a thread and start it running:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

The other way to create a thread is to declare a class that implements the `Runnable` interface. That class then implements the `run` method. An instance of the class can then be allocated, passed as an argument when creating `Thread`, and started. The same example in this other style looks like the following:

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
        . . .
    }
}
```

The following code would then create a thread and start it running:

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

Every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it.

Since:

JDK1.0

See Also:
[Runnable](#), [Runtime.exit\(int\)](#), [run\(\)](#), [stop\(\)](#)

Nested Class Summary

static class	Thread.State A thread state.
static interface	Thread.UncaughtExceptionHandler Interface for handlers invoked when a Thread abruptly terminates due to an uncaught exception.

Field Summary

static int	MAX_PRIORITY The maximum priority that a thread can have.
static int	MIN_PRIORITY The minimum priority that a thread can have.
static int	NORM_PRIORITY The default priority that is assigned to a thread.

Constructor Summary

[Thread\(\)](#)

Allocates a new Thread object.

[Thread\(Runnable target\)](#)

Allocates a new Thread object.

[Thread\(Runnable target, String name\)](#)

Allocates a new Thread object.

[Thread](#)([String](#) name)

Allocates a new Thread object.

[Thread](#)([ThreadGroup](#) group, [Runnable](#) target)

Allocates a new Thread object.

[Thread](#)([ThreadGroup](#) group, [Runnable](#) target, [String](#) name)

Allocates a new Thread object so that it has `target` as its run object, has the specified name as its name, and belongs to the thread group referred to by `group`.

[Thread](#)([ThreadGroup](#) group, [Runnable](#) target, [String](#) name, long `stackSize`)

Allocates a new Thread object so that it has `target` as its run object, has the specified name as its name, belongs to the thread group referred to by `group`, and has the specified *stack size*.

[Thread](#)([ThreadGroup](#) group, [String](#) name)

Allocates a new Thread object.

Method Summary

static int	activeCount () Returns the number of active threads in the current thread's thread group.
void	checkAccess () Determines if the currently running thread has permission to modify this thread.
int	countStackFrames () Deprecated. <i>The definition of this call depends on suspend(), which is deprecated. Further, the results of this call were never well-defined.</i>
static Thread	currentThread () Returns a reference to the currently executing thread object.
void	destroy () Deprecated. <i>If this method were to be implemented, it would be deadlock prone in much the manner of suspend(). If the target thread held a lock on the monitor protecting a critical system resource when it was suspended, no thread could ever access this resource again. If another thread ever attempted to lock this resource, deadlock would result. Such deadlocks typically manifest themselves as "frozen" processes. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.</i>

static void	dumpStack() Prints a stack trace of the current thread.
static int	enumerate() (Thread[] tarray) Copies into the specified array every active thread in the current thread's thread group and its subgroups.
static Map < Thread , StackTraceElement []>	getAllStackTraces() Returns a map of stack traces for all live threads.
ClassLoader	getContextClassLoader() Returns the context ClassLoader for this Thread.
static Thread.UncaughtExceptionHandler	getDefaultUncaughtExceptionHandler() Returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
long	getId() Returns the identifier of this Thread.
String	getName() Returns this thread's name.
int	getPriority() Returns this thread's priority.
StackTraceElement []	getStackTrace() Returns an array of stack trace elements representing the stack dump of this thread.
Thread.State	getState() Returns the state of this thread.
ThreadGroup	getThreadGroup() Returns the thread group to which this thread belongs.
Thread.UncaughtExceptionHandler	getUncaughtExceptionHandler() Returns the handler invoked when this thread abruptly terminates due to an uncaught exception.
static boolean	holdsLock() (Object obj) Returns true if and only if the current thread holds the monitor lock on the specified object.
void	interrupt() Interrupts this thread.
static boolean	interrupted() Tests whether the current thread has been interrupted.

boolean	isAlive() Tests if this thread is alive.
boolean	isDaemon() Tests if this thread is a daemon thread.
boolean	isInterrupted() Tests whether this thread has been interrupted.
void	join() Waits for this thread to die.
void	join(long millis) Waits at most <code>millis</code> milliseconds for this thread to die.
void	join(long millis, int nanos) Waits at most <code>millis</code> milliseconds plus <code>nanos</code> nanoseconds for this thread to die.
void	resume() Deprecated. <i>This method exists solely for use with <code>suspend()</code>, which has been deprecated because it is deadlock-prone. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.</i>
void	run() If this thread was constructed using a separate <code>Runnable</code> run object, then that <code>Runnable</code> object's <code>run</code> method is called; otherwise, this method does nothing and returns.
void	setContextClassLoader(ClassLoader cl) Sets the context <code>ClassLoader</code> for this <code>Thread</code> .
void	setDaemon(boolean on) Marks this thread as either a daemon thread or a user thread.
static void	setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh) Set the default handler invoked when a thread abruptly terminates due to an uncaught exception, and no other handler has been defined for that thread.
void	setName(String name) Changes the name of this thread to be equal to the argument <code>name</code> .
void	setPriority(int newPriority) Changes the priority of this thread.

void	<code>setUncaughtExceptionHandler</code> (<code>Thread.UncaughtExceptionHandler</code> eh) Set the handler invoked when this thread abruptly terminates due to an uncaught exception.
static void	<code>sleep</code> (long millis) Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
static void	<code>sleep</code> (long millis, int nanos) Causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds.
void	<code>start</code> () Causes this thread to begin execution; the Java Virtual Machine calls the <code>run</code> method of this thread.
void	<code>stop</code> () Deprecated. <i>This method is inherently unsafe. Stopping a thread with <code>Thread.stop</code> causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked <code>ThreadDeath</code> exception propagating up the stack). If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior. Many uses of <code>stop</code> should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its <code>run</code> method in an orderly fashion if the variable indicates that it is to stop running. If the target thread waits for long periods (on a condition variable, for example), the <code>interrupt</code> method should be used to interrupt the wait. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.</i>
void	<code>stop</code> (<code>Throwable</code> obj) Deprecated. <i>This method is inherently unsafe. See stop() for details. An additional danger of this method is that it may be used to generate exceptions that the target thread is unprepared to handle (including checked exceptions that the thread could not possibly throw, were it not for this method). For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.</i>

void	suspend () <i>Deprecated. This method has been deprecated, as it is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling <code>resume</code>, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.</i>
String	toString () Returns a string representation of this thread, including the thread's name, priority, and thread group.
static void	yield () Causes the currently executing thread object to temporarily pause and allow other threads to execute.

Methods inherited from class java.lang.[Object](#)

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [wait](#), [wait](#), [wait](#)

Field Detail

MIN_PRIORITY

```
public static final int MIN_PRIORITY
```

The minimum priority that a thread can have.

See Also:

[Constant Field Values](#)

NORM_PRIORITY

```
public static final int NORM_PRIORITY
```

The default priority that is assigned to a thread.

See Also:[Constant Field Values](#)

MAX_PRIORITY

```
public static final int MAX_PRIORITY
```

The maximum priority that a thread can have.

See Also:[Constant Field Values](#)

Constructor Detail

Thread

```
public Thread()
```

Allocates a new Thread object. This constructor has the same effect as `Thread(null, null, gname)`, where ***gname*** is a newly generated name. Automatically generated names are of the form "Thread- "+*n*, where *n* is an integer.

See Also:[Thread\(java.lang.ThreadGroup, java.lang.Runnable, java.lang.String\)](#)

Thread

```
public Thread(Runnable target)
```

Allocates a new Thread object. This constructor has the same effect as `Thread(null, target, gname)`, where *gname* is a newly generated name. Automatically generated names are of the form "Thread- "+*n*, where *n* is an integer.

Parameters:

`target` - the object whose run method is called.

See Also:

[Thread\(java.lang.ThreadGroup, java.lang.Runnable, java.lang.String\)](#)

Thread

```
public Thread(ThreadGroup group,  
             Runnable target)
```

Allocates a new Thread object. This constructor has the same effect as Thread(group, target, *gname*), where *gname* is a newly generated name. Automatically generated names are of the form "Thread - "+*n*, where *n* is an integer.

Parameters:

group - the thread group.

target - the object whose run method is called.

Throws:

[SecurityException](#) - if the current thread cannot create a thread in the specified thread group.

See Also:

[Thread\(java.lang.ThreadGroup, java.lang.Runnable, java.lang.String\)](#)

Thread

```
public Thread(String name)
```

Allocates a new Thread object. This constructor has the same effect as Thread(null, null, name).

Parameters:

name - the name of the new thread.

See Also:

[Thread\(java.lang.ThreadGroup, java.lang.Runnable, java.lang.String\)](#)

Thread

```
public Thread(ThreadGroup group,  
             String name)
```

Allocates a new Thread object. This constructor has the same effect as Thread(group, null, name)

Parameters:

group - the thread group.

name - the name of the new thread.

Throws:

[SecurityException](#) - if the current thread cannot create a thread in the specified thread group.

See Also:

[Thread\(java.lang.ThreadGroup, java.lang.Runnable, java.lang.String\)](#)

Thread

```
public Thread(Runnable target,  
             String name)
```

Allocates a new Thread object. This constructor has the same effect as Thread(null, target, name).

Parameters:

target - the object whose run method is called.

name - the name of the new thread.

See Also:

[Thread\(java.lang.ThreadGroup, java.lang.Runnable, java.lang.String\)](#)

Thread

```
public Thread(ThreadGroup group,  
             Runnable target,  
             String name)
```

Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.

If `group` is `null` and there is a security manager, the group is determined by the security manager's `getThreadGroup` method. If `group` is `null` and there is not a security manager, or the security manager's `getThreadGroup` method returns `null`, the group is set to be the same `ThreadGroup` as the thread that is creating the new thread.

If there is a security manager, its `checkAccess` method is called with the `ThreadGroup` as its argument.

In addition, its `checkPermission` method is called with the `RuntimePermission("enableContextClassLoaderOverride")` permission when invoked directly or indirectly by the constructor of a subclass which overrides the `getContextClassLoader` or `setContextClassLoader` methods. This may result in a `SecurityException`.

If the `target` argument is not `null`, the `run` method of the `target` is called when this thread is started. If the `target` argument is `null`, this thread's `run` method is called when this thread is started.

The priority of the newly created thread is set equal to the priority of the thread creating it, that is, the currently running thread. The method `setPriority` may be used to change the priority to a new value.

The newly created thread is initially marked as being a daemon thread if and only if the thread creating it is currently marked as a daemon thread. The method `setDaemon` may be used to change whether or not a thread is a daemon.

Parameters:

- `group` - the thread group.
- `target` - the object whose `run` method is called.
- `name` - the name of the new thread.

Throws:

[SecurityException](#) - if the current thread cannot create a thread in the specified thread group or cannot override the context class loader methods.

See Also:

[Runnable.run\(\)](#), [run\(\)](#), [setDaemon\(boolean\)](#), [setPriority\(int\)](#), [ThreadGroup.checkAccess\(\)](#), [SecurityManager.checkAccess\(java.lang.Thread\)](#)

Thread

```
public Thread(ThreadGroup group,
              Runnable target,
```

[String](#) name,
long stackSize)

Allocates a new `Thread` object so that it has `target` as its run object, has the specified `name` as its name, belongs to the thread group referred to by `group`, and has the specified *stack size*.

This constructor is identical to [Thread\(ThreadGroup, Runnable, String\)](#) with the exception of the fact that it allows the thread stack size to be specified. The stack size is the approximate number of bytes of address space that the virtual machine is to allocate for this thread's stack. **The effect of the `stackSize` parameter, if any, is highly platform dependent.**

On some platforms, specifying a higher value for the `stackSize` parameter may allow a thread to achieve greater recursion depth before throwing a [StackOverflowError](#). Similarly, specifying a lower value may allow a greater number of threads to exist concurrently without throwing an [OutOfMemoryError](#) (or other internal error). The details of the relationship between the value of the `stackSize` parameter and the maximum recursion depth and concurrency level are platform-dependent. **On some platforms, the value of the `stackSize` parameter may have no effect whatsoever.**

The virtual machine is free to treat the `stackSize` parameter as a suggestion. If the specified value is unreasonably low for the platform, the virtual machine may instead use some platform-specific minimum value; if the specified value is unreasonably high, the virtual machine may instead use some platform-specific maximum. Likewise, the virtual machine is free to round the specified value up or down as it sees fit (or to ignore it completely).

Specifying a value of zero for the `stackSize` parameter will cause this constructor to behave exactly like the `Thread(ThreadGroup, Runnable, String)` constructor.

Due to the platform-dependent nature of the behavior of this constructor, extreme care should be exercised in its use. The thread stack size necessary to perform a given computation will likely vary from one JRE implementation to another. In light of this variation, careful tuning of the stack size parameter may be required, and the tuning may need to be repeated for each JRE implementation on which an application is to run.

Implementation note: Java platform implementers are encouraged to document their implementation's behavior with respect to the `stackSize` parameter.

Parameters:

`group` - the thread group.

`target` - the object whose `run` method is called.

`name` - the name of the new thread.

`stackSize` - the desired stack size for the new thread, or zero to indicate that this parameter is to be ignored.

Throws:

[SecurityException](#) - if the current thread cannot create a thread in the specified thread group.

Method Detail

currentThread

```
public static Thread currentThread()
```

Returns a reference to the currently executing thread object.

Returns:

the currently executing thread.

yield

```
public static void yield()
```

Causes the currently executing thread object to temporarily pause and allow other threads to execute.

sleep

```
public static void sleep(long millis)  
    throws InterruptedException
```

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. The thread does not lose ownership of any monitors.

Parameters:

`millis` - the length of time to sleep in milliseconds.

Throws:

[InterruptedException](#) - if another thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

See Also:

[Object.notify\(\)](#)

sleep

```
public static void sleep(long millis,  
                        int nanos)  
    throws InterruptedException
```

Causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds. The thread does not lose ownership of any monitors.

Parameters:

`millis` - the length of time to sleep in milliseconds.

`nanos` - 0-999999 additional nanoseconds to sleep.

Throws:

[IllegalArgumentException](#) - if the value of `millis` is negative or the value of `nanos` is not in the range 0-999999.

[InterruptedException](#) - if another thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

See Also:

[Object.notify\(\)](#)

start

```
public void start()
```

Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread.

The result is that two threads are running concurrently: the current thread (which returns from the call to the `start` method) and the other thread (which executes its `run` method).

It is never legal to start a thread more than once. In particular, a thread may not be restarted once it has completed execution.

Throws:

[IllegalThreadStateException](#) - if the thread was already started.

See Also:

[run\(\)](#), [stop\(\)](#)

run

```
public void run()
```

If this thread was constructed using a separate `Runnable` run object, then that `Runnable` object's `run` method is called; otherwise, this method does nothing and returns.

Subclasses of `Thread` should override this method.

Specified by:

[run](#) in interface [Runnable](#)

See Also:

[start\(\)](#), [stop\(\)](#), [Thread\(java.lang.ThreadGroup, java.lang.Runnable, java.lang.String\)](#), [Runnable.run\(\)](#)

stop

```
public final void stop()
```

Deprecated. *This method is inherently unsafe. Stopping a thread with `Thread.stop` causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked `ThreadDeath` exception propagating up the stack). If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior. Many uses of `stop` should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its `run` method in an orderly fashion if the variable indicates that it is to stop running. If the target thread waits for long periods (on a condition variable, for example), the `interrupt` method should be used to interrupt the wait. For more information, see [Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?](#).*

Forces the thread to stop executing.

If there is a security manager installed, its `checkAccess` method is called with `this` as its argument. This may result in a `SecurityException` being raised (in the current thread).

If this thread is different from the current thread (that is, the current thread is trying to stop a thread other than itself), the security manager's `checkPermission` method (with a `RuntimePermission("stopThread")` argument) is called in addition. Again, this may result in throwing a `SecurityException` (in the current thread).

The thread represented by this thread is forced to stop whatever it is doing abnormally and to throw a newly created `ThreadDeath` object as an exception.

It is permitted to stop a thread that has not yet been started. If the thread is eventually started, it immediately terminates.

An application should not normally try to catch `ThreadDeath` unless it must do some extraordinary cleanup operation (note that the throwing of `ThreadDeath` causes `finally` clauses of `try` statements to be executed before the thread officially dies). If a `catch` clause catches a `ThreadDeath` object, it is important to rethrow the object so that the thread actually dies.

The top-level error handler that reacts to otherwise uncaught exceptions does not print out a message or otherwise notify the application if the uncaught exception is an instance of `ThreadDeath`.

Throws:

[SecurityException](#) - if the current thread cannot modify this thread.

See Also:

[interrupt\(\)](#), [checkAccess\(\)](#), [run\(\)](#), [start\(\)](#), [ThreadDeath](#), [ThreadGroup.uncaughtException\(java.lang.Thread, java.lang.Throwable\)](#), [SecurityManager.checkAccess\(Thread\)](#), [SecurityManager.checkPermission\(java.security.Permission\)](#)

stop

```
public final void stop(Throwable obj)
```

Deprecated. *This method is inherently unsafe. See [stop\(\)](#) for details. An additional danger of this method is that it may be used to generate exceptions that the target thread is unprepared to handle (including checked exceptions that the thread could not possibly throw, were it not for this method). For more information, see [Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?](#).*

Forces the thread to stop executing.

If there is a security manager installed, the `checkAccess` method of this thread is called, which may result in a `SecurityException` being raised (in the current thread).

If this thread is different from the current thread (that is, the current thread is trying to stop a thread other than itself) or `obj` is not an instance of `ThreadDeath`, the security manager's

`checkPermission` method (with the `RuntimePermission("stopThread")` argument) is called in addition. Again, this may result in throwing a `SecurityException` (in the current thread).

If the argument `obj` is null, a `NullPointerException` is thrown (in the current thread).

The thread represented by this thread is forced to complete whatever it is doing abnormally and to throw the `Throwable` object `obj` as an exception. This is an unusual action to take; normally, the `stop` method that takes no arguments should be used.

It is permitted to stop a thread that has not yet been started. If the thread is eventually started, it immediately terminates.

Parameters:

`obj` - the `Throwable` object to be thrown.

Throws:

[SecurityException](#) - if the current thread cannot modify this thread.

See Also:

[interrupt\(\)](#), [checkAccess\(\)](#), [run\(\)](#), [start\(\)](#), [stop\(\)](#),
[SecurityManager.checkAccess\(Thread\)](#), [SecurityManager.
 checkPermission\(java.security.Permission\)](#)

interrupt

```
public void interrupt()
```

Interrupts this thread.

Unless the current thread is interrupting itself, which is always permitted, the [checkAccess](#) method of this thread is invoked, which may cause a [SecurityException](#) to be thrown.

If this thread is blocked in an invocation of the [wait\(\)](#), [wait\(long\)](#), or [wait\(long, int\)](#) methods of the [Object](#) class, or of the [join\(\)](#), [join\(long\)](#), [join\(long, int\)](#), [sleep\(long\)](#), or [sleep\(long, int\)](#), methods of this class, then its interrupt status will be cleared and it will receive an [InterruptedException](#).

If this thread is blocked in an I/O operation upon an [interruptible channel](#) then the channel will be closed, the thread's interrupt status will be set, and the thread will receive a [ClosedByInterruptException](#).

If this thread is blocked in a [Selector](#) then the thread's interrupt status will be set and it will

return immediately from the selection operation, possibly with a non-zero value, just as if the selector's [wakeUp](#) method were invoked.

If none of the previous conditions hold then this thread's interrupt status will be set.

Throws:

[SecurityException](#) - if the current thread cannot modify this thread

interrupted

```
public static boolean interrupted()
```

Tests whether the current thread has been interrupted. The *interrupted status* of the thread is cleared by this method. In other words, if this method were to be called twice in succession, the second call would return false (unless the current thread were interrupted again, after the first call had cleared its interrupted status and before the second call had examined it).

Returns:

true if the current thread has been interrupted; false otherwise.

See Also:

[isInterrupted\(\)](#)

isInterrupted

```
public boolean isInterrupted()
```

Tests whether this thread has been interrupted. The *interrupted status* of the thread is unaffected by this method.

Returns:

true if this thread has been interrupted; false otherwise.

See Also:

[interrupted\(\)](#)

destroy

```
public void destroy()
```

Deprecated. *If this method were to be implemented, it would be deadlock prone in much the manner of [suspend\(\)](#). If the target thread held a lock on the monitor protecting a critical system resource when it was suspended, no thread could ever access this resource again. If another thread ever attempted to lock this resource, deadlock would result. Such deadlocks typically manifest themselves as "frozen" processes. For more information, see [Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?](#).*

Destroys this thread, without any cleanup. Any monitors it has locked remain locked. (This method is not implemented.)

isAlive

```
public final boolean isAlive()
```

Tests if this thread is alive. A thread is alive if it has been started and has not yet died.

Returns:

true if this thread is alive; false otherwise.

suspend

```
public final void suspend()
```

Deprecated. *This method has been deprecated, as it is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling `resume`, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes. For more information, see [Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?](#).*

Suspends this thread.

First, the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException` (in the current thread).

If the thread is alive, it is suspended and makes no further progress unless and until it is resumed.

Throws:

[SecurityException](#) - if the current thread cannot modify this thread.

See Also:

[checkAccess\(\)](#)

resume

```
public final void resume()
```

Deprecated. *This method exists solely for use with [suspend\(\)](#), which has been deprecated because it is deadlock-prone. For more information, see [Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?](#).*

Resumes a suspended thread.

First, the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException` (in the current thread).

If the thread is alive but suspended, it is resumed and is permitted to make progress in its execution.

Throws:

[SecurityException](#) - if the current thread cannot modify this thread.

See Also:

[checkAccess\(\)](#), [suspend\(\)](#)

setPriority

```
public final void setPriority(int newPriority)
```

Changes the priority of this thread.

First the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException`.

Otherwise, the priority of this thread is set to the smaller of the specified `newPriority` and the maximum permitted priority of the thread's thread group.

Parameters:

`newPriority` - priority to set this thread to

Throws:

[IllegalArgumentException](#) - If the priority is not in the range

MIN_PRIORITY to MAX_PRIORITY.

[SecurityException](#) - if the current thread cannot modify this thread.

See Also:

[getPriority\(\)](#), [checkAccess\(\)](#), [getPriority\(\)](#), [getThreadGroup\(\)](#),
[MAX_PRIORITY](#), [MIN_PRIORITY](#), [ThreadGroup.getMaxPriority\(\)](#)

getPriority

```
public final int getPriority()
```

Returns this thread's priority.

Returns:

this thread's priority.

See Also:

[setPriority\(int\)](#), [setPriority\(int\)](#)

setName

```
public final void setName(String name)
```

Changes the name of this thread to be equal to the argument name.

First the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException`.

Parameters:

name - the new name for this thread.

Throws:

[SecurityException](#) - if the current thread cannot modify this thread.

See Also:

[getName\(\)](#), [checkAccess\(\)](#), [getName\(\)](#)

getName

```
public final String getName()
```

Returns this thread's name.

Returns:

this thread's name.

See Also:

[setName\(java.lang.String\)](#), [setName\(java.lang.String\)](#)

getThreadGroup

```
public final ThreadGroup getThreadGroup()
```

Returns the thread group to which this thread belongs. This method returns null if this thread has died (been stopped).

Returns:

this thread's thread group.

activeCount

```
public static int activeCount()
```

Returns the number of active threads in the current thread's thread group.

Returns:

the number of active threads in the current thread's thread group.

enumerate

```
public static int enumerate(Thread[] tarray)
```

Copies into the specified array every active thread in the current thread's thread group and its subgroups. This method simply calls the `enumerate` method of the current thread's thread group with the array argument.

First, if there is a security manager, that `enumerate` method calls the security manager's

`checkAccess` method with the thread group as its argument. This may result in throwing a `SecurityException`.

Parameters:

`tarray` - an array of `Thread` objects to copy to

Returns:

the number of threads put into the array

Throws:

[SecurityException](#) - if a security manager exists and its `checkAccess` method doesn't allow the operation.

See Also:

[ThreadGroup.enumerate\(java.lang.Thread\[\]\)](#), [SecurityManager.checkAccess\(java.lang.ThreadGroup\)](#)

countStackFrames

```
public int countStackFrames()
```

Deprecated. *The definition of this call depends on [suspend\(\)](#), which is deprecated. Further, the results of this call were never well-defined.*

Counts the number of stack frames in this thread. The thread must be suspended.

Returns:

the number of stack frames in this thread.

Throws:

[IllegalThreadStateException](#) - if this thread is not suspended.

join

```
public final void join(long millis)
    throws InterruptedException
```

Waits at most `millis` milliseconds for this thread to die. A timeout of 0 means to wait forever.

Parameters:

`millis` - the time to wait in milliseconds.

Throws:

[InterruptedException](#) - if another thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

join

```
public final void join(long millis,  
                        int nanos)  
    throws InterruptedException
```

Waits at most `millis` milliseconds plus `nanos` nanoseconds for this thread to die.

Parameters:

`millis` - the time to wait in milliseconds.

`nanos` - 0-999999 additional nanoseconds to wait.

Throws:

[IllegalArgumentException](#) - if the value of `millis` is negative the value of `nanos` is not in the range 0-999999.

[InterruptedException](#) - if another thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

join

```
public final void join()  
    throws InterruptedException
```

Waits for this thread to die.

Throws:

[InterruptedException](#) - if another thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

dumpStack

```
public static void dumpStack()
```

Prints a stack trace of the current thread. This method is used only for debugging.

See Also:[Throwable.printStackTrace\(\)](#)

setDaemon

```
public final void setDaemon(boolean on)
```

Marks this thread as either a daemon thread or a user thread. The Java Virtual Machine exits when the only threads running are all daemon threads.

This method must be called before the thread is started.

This method first calls the `checkAccess` method of this thread with no arguments. This may result in throwing a `SecurityException` (in the current thread).

Parameters:

`on` - if `true`, marks this thread as a daemon thread.

Throws:

[IllegalThreadStateException](#) - if this thread is active.

[SecurityException](#) - if the current thread cannot modify this thread.

See Also:

[isDaemon\(\)](#), [checkAccess\(\)](#)

isDaemon

```
public final boolean isDaemon()
```

Tests if this thread is a daemon thread.

Returns:

`true` if this thread is a daemon thread; `false` otherwise.

See Also:

[setDaemon\(boolean\)](#)

checkAccess

```
public final void checkAccess()
```

Determines if the currently running thread has permission to modify this thread.

If there is a security manager, its `checkAccess` method is called with this thread as its argument. This may result in throwing a `SecurityException`.

Note: This method was mistakenly non-final in JDK 1.1. It has been made final in the Java 2 Platform.

Throws:

[SecurityException](#) - if the current thread is not allowed to access this thread.

See Also:

[SecurityManager.checkAccess\(java.lang.Thread\)](#)

toString

```
public String toString()
```

Returns a string representation of this thread, including the thread's name, priority, and thread group.

Overrides:

[toString](#) in class [Object](#)

Returns:

a string representation of this thread.

getContextClassLoader

```
public ClassLoader getContextClassLoader()
```

Returns the context `ClassLoader` for this `Thread`. The context `ClassLoader` is provided by the creator of the thread for use by code running in this thread when loading classes and resources. If not set, the default is the `ClassLoader` context of the parent `Thread`. The context `ClassLoader` of the primordial thread is typically set to the class loader used to load the application.

First, if there is a security manager, and the caller's class loader is not null and the caller's class loader is not the same as or an ancestor of the context class loader for the thread whose context class loader is being requested, then the security manager's `checkPermission` method is called with a `RuntimePermission("getClassLoader")` permission to see

if it's ok to get the context ClassLoader..

Returns:

the context ClassLoader for this Thread

Throws:

[SecurityException](#) - if a security manager exists and its `checkPermission` method doesn't allow getting the context ClassLoader.

Since:

1.2

See Also:

[setContextClassLoader\(java.lang.ClassLoader\)](#),
[SecurityManager.checkPermission\(java.security.Permission\)](#),
[RuntimePermission](#)

setContextClassLoader

```
public void setContextClassLoader(ClassLoader cl)
```

Sets the context ClassLoader for this Thread. The context ClassLoader can be set when a thread is created, and allows the creator of the thread to provide the appropriate class loader to code running in the thread when loading classes and resources.

First, if there is a security manager, its `checkPermission` method is called with a `RuntimePermission("setContextClassLoader")` permission to see if it's ok to set the context ClassLoader..

Parameters:

`cl` - the context ClassLoader for this Thread

Throws:

[SecurityException](#) - if the current thread cannot set the context ClassLoader.

Since:

1.2

See Also:

[getContextClassLoader\(\)](#), [SecurityManager.checkPermission\(java.security.Permission\)](#), [RuntimePermission](#)

holdsLock

```
public static boolean holdsLock(Object obj)
```

Returns `true` if and only if the current thread holds the monitor lock on the specified object.

This method is designed to allow a program to assert that the current thread already holds a specified lock:

```
assert Thread.holdsLock(obj);
```

Parameters:

`obj` - the object on which to test lock ownership

Returns:

`true` if the current thread holds the monitor lock on the specified object.

Throws:

[NullPointerException](#) - if `obj` is `null`

Since:

1.4

getStackTrace

```
public StackTraceElement[] getStackTrace()
```

Returns an array of stack trace elements representing the stack dump of this thread. This method will return a zero-length array if this thread has not started or has terminated. If the returned array is of non-zero length then the first element of the array represents the top of the stack, which is the most recent method invocation in the sequence. The last element of the array represents the bottom of the stack, which is the least recent method invocation in the sequence.

If there is a security manager, and this thread is not the current thread, then the security manager's `checkPermission` method is called with a `RuntimePermission` ("getStackTrace") permission to see if it's ok to get the stack trace.

Some virtual machines may, under some circumstances, omit one or more stack frames from the stack trace. In the extreme case, a virtual machine that has no stack trace information concerning this thread is permitted to return a zero-length array from this method.

Returns:

an array of `StackTraceElement`, each represents one stack frame.

Throws:

[SecurityException](#) - if a security manager exists and its `checkPermission` method doesn't allow getting the stack trace of thread.

Since:

1.5

See Also:

[SecurityManager.checkPermission\(java.security.Permission\)](#),
[RuntimePermission](#), [Throwable.getStackTrace\(\)](#)

getAllStackTraces

```
public static Map<Thread, StackTraceElement[]> getAllStackTraces()
```

Returns a map of stack traces for all live threads. The map keys are threads and each map value is an array of `StackTraceElement` that represents the stack dump of the corresponding `Thread`. The returned stack traces are in the format specified for the [getStackTrace](#) method.

The threads may be executing while this method is called. The stack trace of each thread only represents a snapshot and each stack trace may be obtained at different time. A zero-length array will be returned in the map value if the virtual machine has no stack trace information about a thread.

If there is a security manager, then the security manager's `checkPermission` method is called with a `RuntimePermission("getStackTrace")` permission as well as `RuntimePermission("modifyThreadGroup")` permission to see if it is ok to get the stack trace of all threads.

Returns:

a `Map` from `Thread` to an array of `StackTraceElement` that represents the stack trace of the corresponding thread.

Throws:

[SecurityException](#) - if a security manager exists and its `checkPermission` method doesn't allow getting the stack trace of thread.

Since:

1.5

See Also:

[getStackTrace\(\)](#), [SecurityManager.checkPermission\(java.security.Permission\)](#), [RuntimePermission](#), [Throwable.getStackTrace\(\)](#)

getId

```
public long getId()
```

Returns the identifier of this Thread. The thread ID is a positive `long` number generated when this thread was created. The thread ID is unique and remains unchanged during its lifetime. When a thread is terminated, this thread ID may be reused.

Returns:

this thread's ID.

Since:

1.5

getState

```
public Thread.State getState()
```

Returns the state of this thread. This method is designed for use in monitoring of the system state, not for synchronization control.

Returns:

this thread's state.

Since:

1.5

setDefaultUncaughtExceptionHandler

```
public static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)
```

Set the default handler invoked when a thread abruptly terminates due to an uncaught exception, and no other handler has been defined for that thread.

Uncaught exception handling is controlled first by the thread, then by the thread's [ThreadGroup](#) object and finally by the default uncaught exception handler. If the thread does not have an explicit uncaught exception handler set, and the thread's thread group (including parent thread groups) does not specialize its `uncaughtException` method, then the default handler's `uncaughtException` method will be invoked.

By setting the default uncaught exception handler, an application can change the way in which uncaught exceptions are handled (such as logging to a specific device, or file) for those threads that would already accept whatever "default" behavior the system provided.

Note that the default uncaught exception handler should not usually defer to the thread's `ThreadGroup` object, as that could cause infinite recursion.

Parameters:

`eh` - the object to use as the default uncaught exception handler. If `null` then there is no default handler.

Throws:

[SecurityException](#) - if a security manager is present and it denies
[RuntimePermission](#) ("setDefaultUncaughtExceptionHandler")

Since:

1.5

See Also:

[setUncaughtExceptionHandler\(java.lang.Thread.UncaughtExceptionHandler\)](#), [getUncaughtExceptionHandler\(\)](#), [ThreadGroup.uncaughtException\(java.lang.Thread, java.lang.Throwable\)](#)

getDefaultUncaughtExceptionHandler

```
public static Thread.UncaughtExceptionHandler
getDefaultUncaughtExceptionHandler()
```

Returns the default handler invoked when a thread abruptly terminates due to an uncaught exception. If the returned value is `null`, there is no default.

Since:

1.5

See Also:

[setDefaultUncaughtExceptionHandler\(java.lang.Thread.UncaughtExceptionHandler\)](#)

getUncaughtExceptionHandler

```
public Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()
```

Returns the handler invoked when this thread abruptly terminates due to an uncaught exception. If this thread has not had an uncaught exception handler explicitly set then this thread's `ThreadGroup` object is returned, unless this thread has terminated, in which case `null` is returned.

Since:

1.5

setUncaughtExceptionHandler

```
public void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)
```

Set the handler invoked when this thread abruptly terminates due to an uncaught exception.

A thread can take full control of how it responds to uncaught exceptions by having its uncaught exception handler explicitly set. If no such handler is set then the thread's `ThreadGroup` object acts as its handler.

Parameters:

eh - the object to use as this thread's uncaught exception handler. If `null` then this thread has no explicit handler.

Throws:

[SecurityException](#) - if the current thread is not allowed to modify this thread.

Since:

1.5

See Also:

[setDefaultUncaughtExceptionHandler\(java.lang.Thread.UncaughtExceptionHandler\)](#), [ThreadGroup.uncaughtException\(java.lang.Thread, java.lang.Throwable\)](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.5.0*

[PREV CLASS](#) [NEXT CLASS](#)

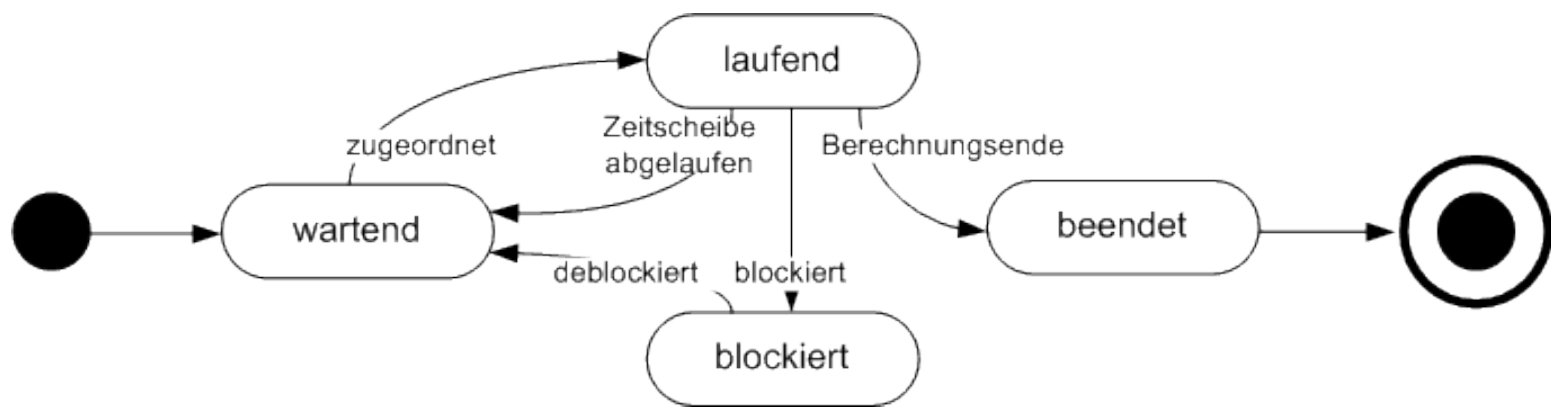
[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).



NAME

pause - suspend the thread until signal is received

SYNOPSIS

```
#include <unistd.h>
```

```
int pause(void);
```

DESCRIPTION

The *pause()* function suspends the calling thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

If the action is to terminate the process, *pause()* will not return.

If the action is to execute a signal-catching function, *pause()* will return after the signal-catching function returns.

RETURN VALUE

Since *pause()* suspends thread execution indefinitely unless interrupted by a signal, there is no successful completion return value. A value of -1 is returned and *errno* is set to indicate the error.

ERRORS

The *pause()* function will fail if:

[EINTR]

A signal is caught by the calling process and control is returned from the signal-catching function.

EXAMPLES

None.

APPLICATION USAGE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[*sigsuspend\(\)*](#), [*<unistd.h>*](#).

DERIVATION

Derived from Issue 1 of the SVID.

UNIX ® is a registered Trademark of The Open Group.
Copyright © 1997 The Open Group
[[Main Index](#) | [XSH](#) | [XCU](#) | [XBD](#) | [XCURSES](#) | [XNS](#)]

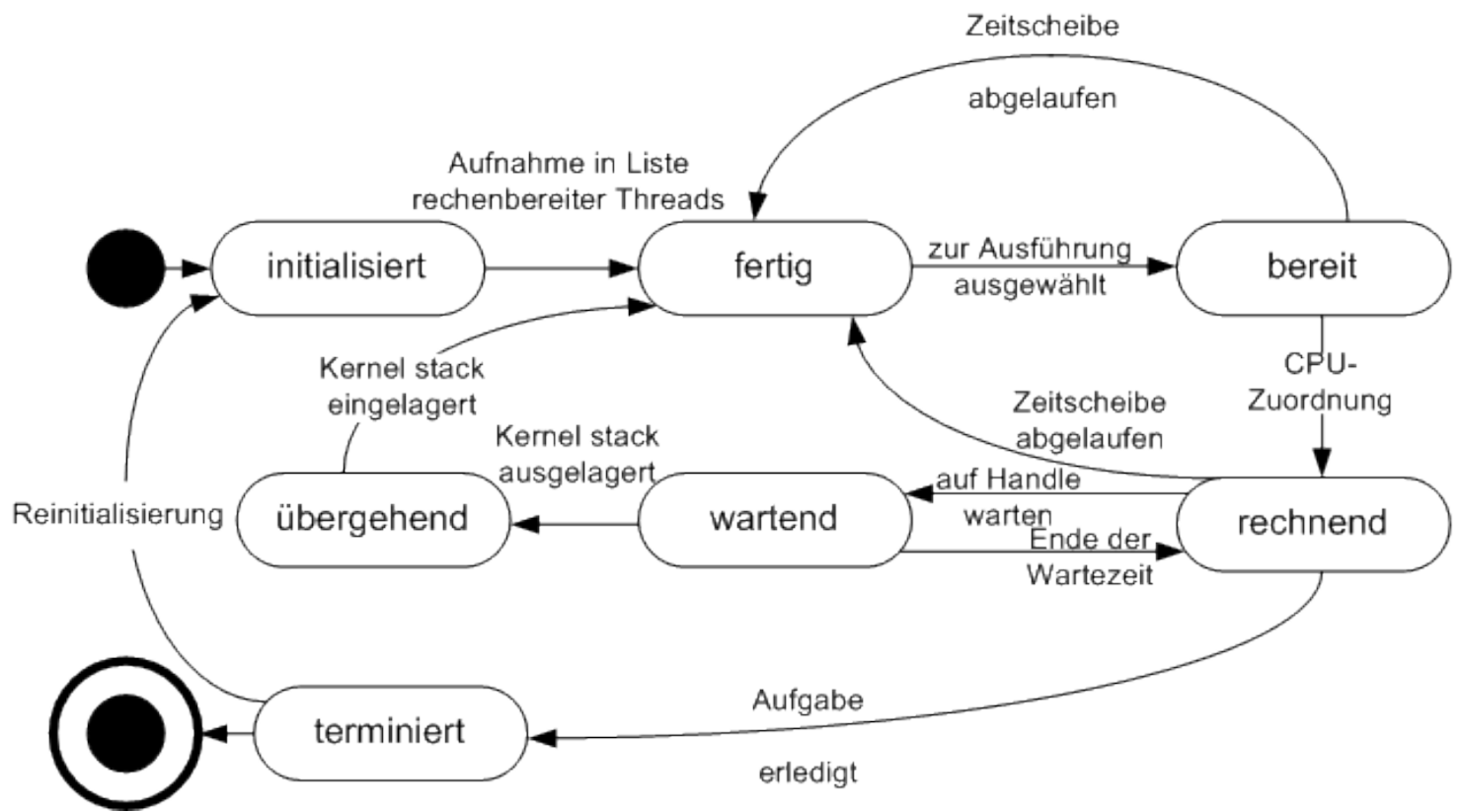
```
#include <signal.h>

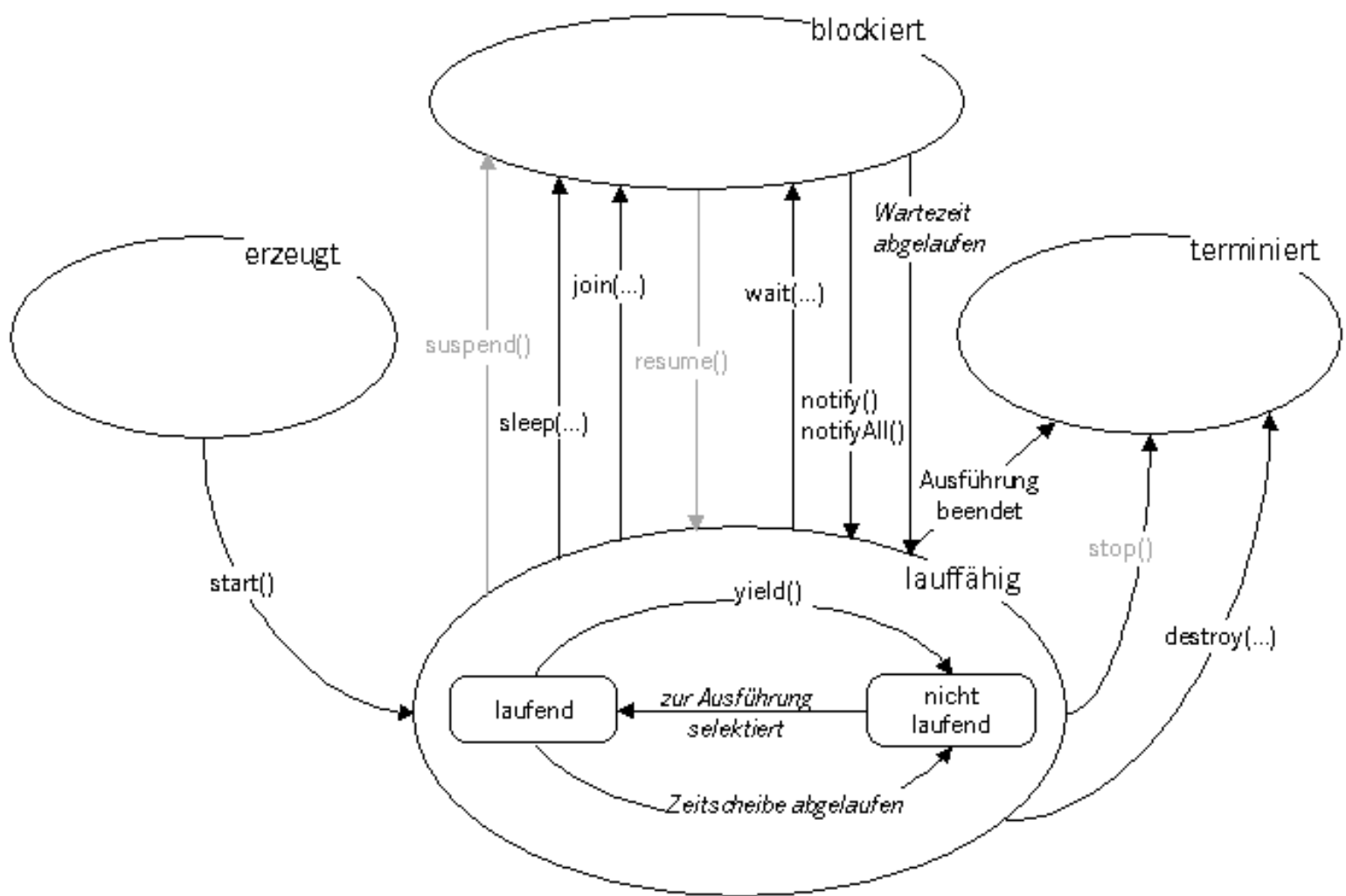
void catcher() {
    printf("SIGUSR1 caught\n");
}

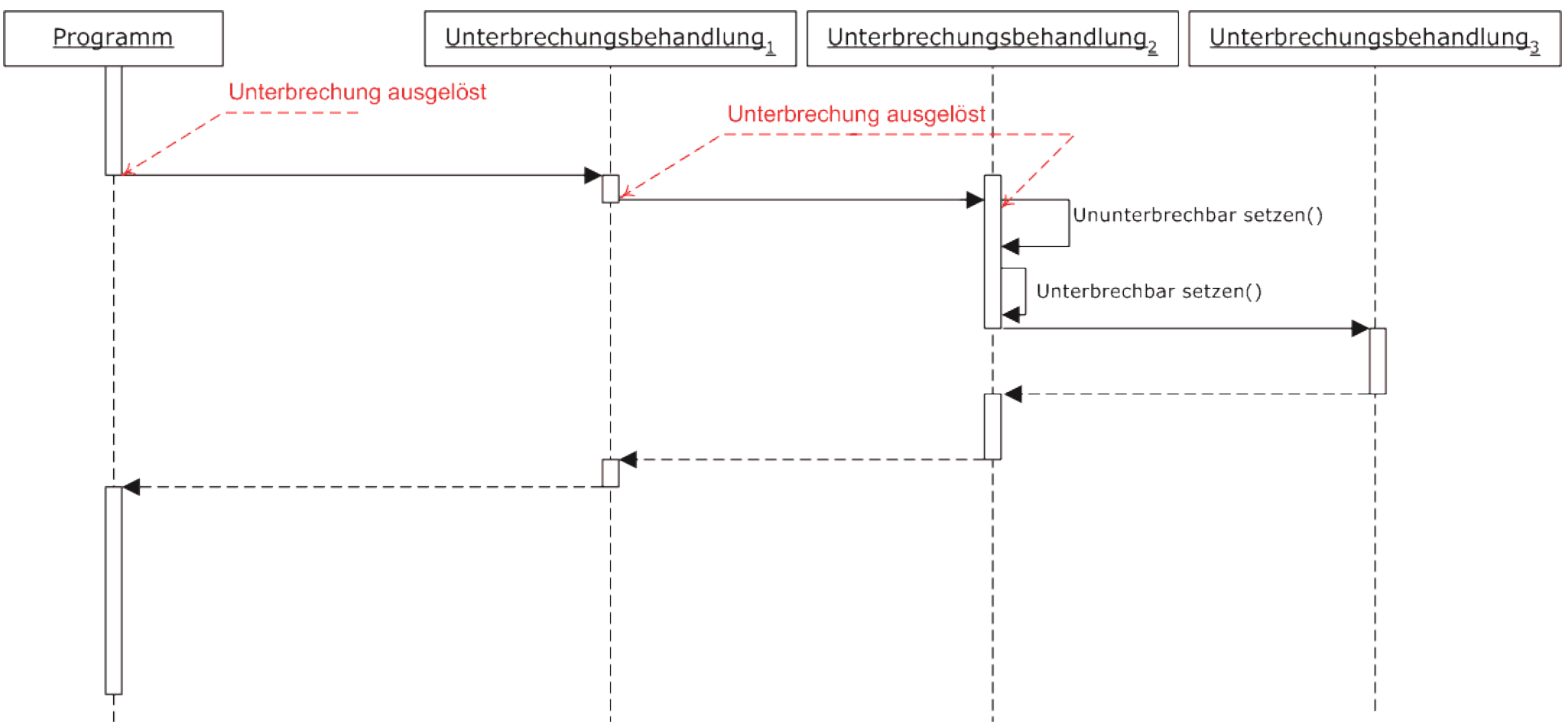
int main(int argc, char** argv[]) {

    signal(SIGUSR1, catcher);

    printf("before\n");
    pause();
    printf("after\n");
}
```







Echtzeit

aus Wikipedia, der freien Enzyklopädie

Echtzeit (engl. *real-time*, dt. veraltet *Realzeit*) ist ein Begriff aus der Informatik.

Die Definition von **Echtzeit** ist umstritten. Die breiteste Zustimmung findet folgende Definition:

Echtzeit bedeutet, dass das Ergebnis einer Berechnung innerhalb eines gewissen Zeitraumes garantiert vorliegt, d.h. bevor eine bestimmte Zeitschranke erreicht ist.

Ein **Echtzeit-System** (engl. *real-time system*) muss also nicht nur ein Berechnungsergebnis mit dem richtigen Wert, sondern dasselbe auch noch **rechtzeitig** liefern. Andernfalls hat das System versagt.

Beispiel: In einem Auto muss das elektronische Motormanagement zu bestimmten Zeitpunkten seine Ergebnisse (einzuspritzende Benzinmenge, Zündzeitpunkt) liefern. Später eintreffende Ergebnisse sind wertlos.

Eine genaue Dauer der Zeitschranke bzw. was rechtzeitig bedeutet, hängt von der Anwendung ab und kann daher nicht allgemein angegeben werden.

Beispiel: In Maschinen ändert sich die gegenseitige Lage von Werkstück und Werkzeug innerhalb von einigen Millisekunden. Abhängig von der geforderten Toleranz muss eine Steuerung innerhalb von einigen hundert Mikro- bis einer Millisekunde auf Abweichungen reagieren, um echtzeit-fähig zu sein. Die Temperatur eines Apparates in einer verfahrenstechnischen Anlage ändert sich meist nur innerhalb von Minuten. Eine Steuerung, die innerhalb von mehreren Sekunden auf Abweichungen reagiert, kann daher noch als echtzeit-

fähig gelten.

Ein **Echtzeit-System** reagiert also auf alle Ereignisse rechtzeitig und verarbeitet die Daten "schritt haltend" mit dem technischen Prozess. Es wird quasi nicht vom technischen Prozess abgehängt.

Um die Echtzeit-Fähigkeit eines **Echtzeit-Systems** theoretisch nachweisen zu können, müssen die Häufigkeit der externen Ereignisse, die Laufzeit der einzelnen Programmteile und die Zeitschranken bekannt sein.

Rechner zur Steuerung von technischen Einrichtungen oder Prozessen wie Maschinen, verfahrenstechnische Anlagen, Autos, Flugzeuge usw. sind praktisch immer **Echtzeit-Systeme**.

Abhängig von den Folgen wird manchmal zwischen **harter Echtzeit** (engl. *hard real-time*) und **weicher Echtzeit** (engl. *soft real-time*) unterschieden.

Beispiele: Das elektronische Motormanagement muss **harte Echtzeit** erfüllen, sonst stottert der Motor oder das Auto bleibt gar ganz stehen, was sogar Unfälle auslösen könnte. Ein Computer, mit dem man Videos und Filme ohne merkbare Wartezeiten bearbeiten können soll, braucht nur **weiche Echtzeit** zu erfüllen. Dauern einzelne Aktionen länger, erfüllt das System dennoch seinen Zweck.

Ein **Echtzeit-System** erfordert nicht zwingend den Einsatz eines Echtzeitbetriebssystems, ein solches erleichtert aber das Schreiben von echtzeit-fähigen Programmen. Auch in speicherprogrammierbare Steuerungen (SPS) und Prozessleitsystemen (PLS) werden Echtzeitbetriebssysteme eingesetzt, die aber dem Anwender nicht direkt zugänglich sind.

Beispiele für Echtzeit-Systeme

- Navigationscomputer im Flugzeug
- Elektronisches Motormanagement im Auto
- Antiblockiersystem (ABS) im Auto
- Video-/Ton-Decoder oder -Encoder (Codec)
- Steuerungen für Maschinen
- Steuerungen für verfahrenstechnische Anlagen

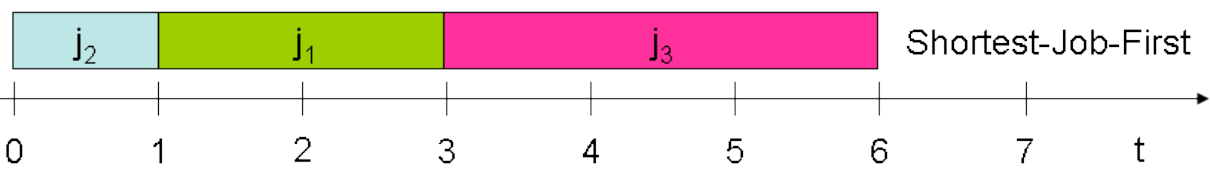
Siehe auch

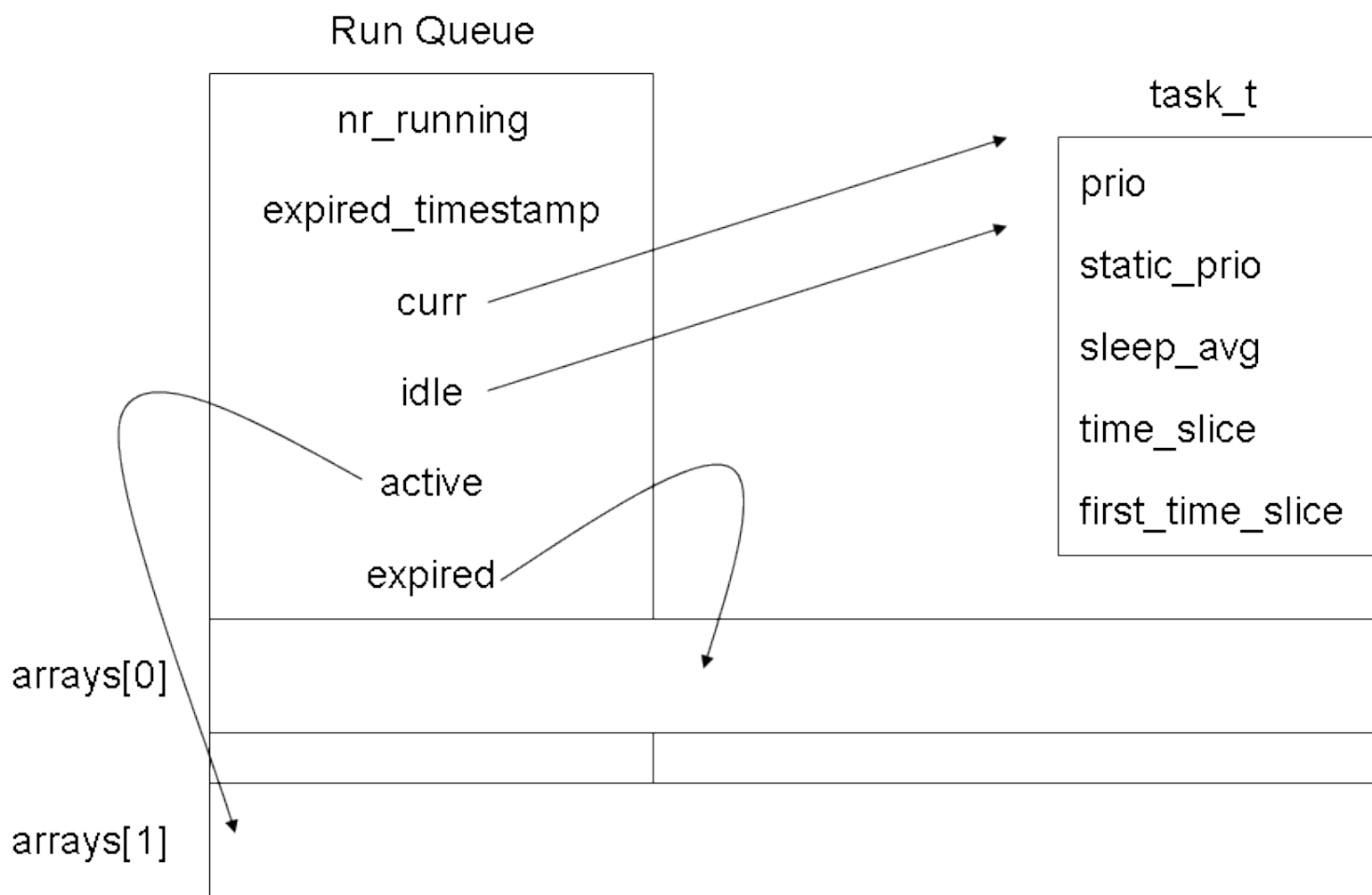
- Echtzeit-System
- Echtzeit-Verarbeitung
- Echtzeit-Programmierung
- Echtzeitbetriebssystem
- Speicherprogrammierbare Steuerung
- Prozessleitsystem
- Echtzeit-Strategiespiel

Weblinks

- Beitrag "Comp.realtime: Frequently Asked Questions (FAQs)", regelmässig in der Newsgroup "comp.realtime" gepostet
- <http://www.faqs.org/faqs/realtime-computing/faq/>
- <http://www.cis.ohio-state.edu/hypertext/faq/usenet/realtime-computing/top.html>

-
- Diese Seite wurde zuletzt geändert um 10:29, 31. Mai 2004.
 - Der Inhalt dieser Seite steht unter der [GNU Free Documentation License](#).





[Home](#) | [Agenda](#) | [Tutorials](#) | [Exercises](#) | [Abstracts](#) | [LC Workshops](#) | [Comments](#) | [Search](#)

[Privacy & Legal Notice](#)

posix threads programming

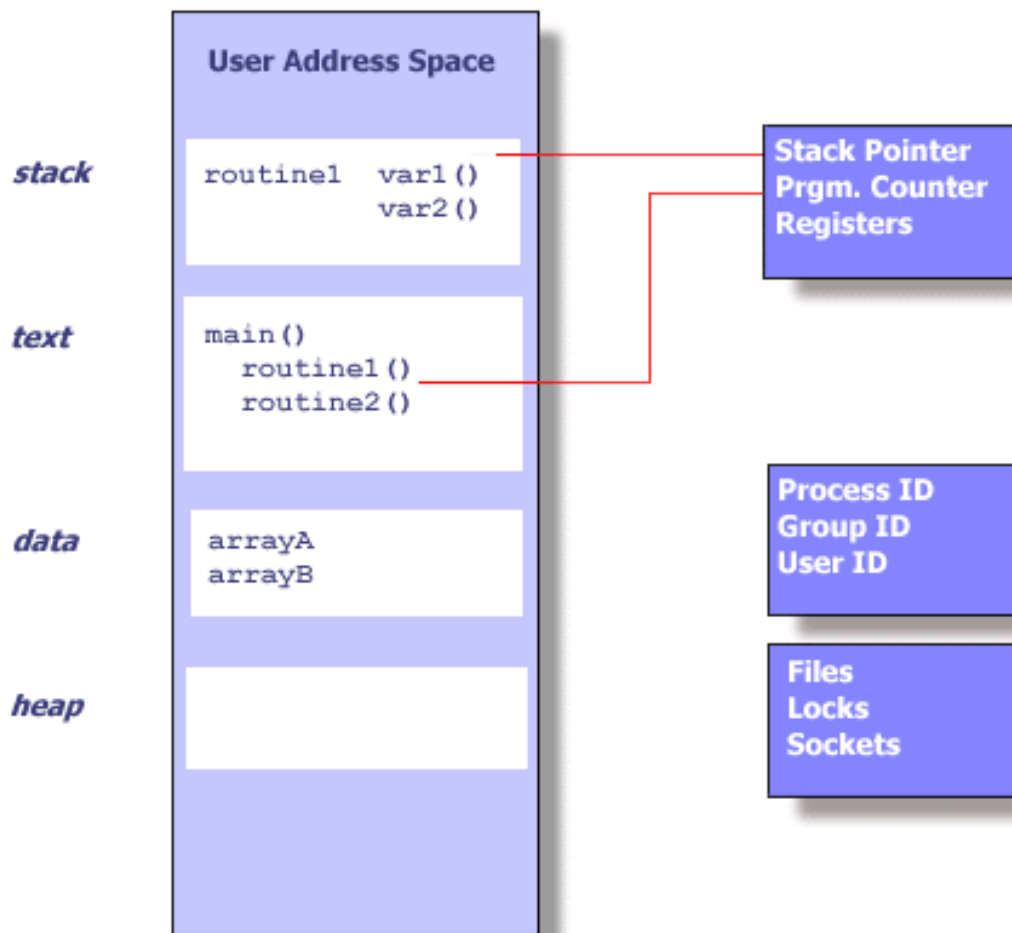
Table of Contents

1. [Pthreads Overview](#)
 1. [What is a Thread?](#)
 2. [What are Pthreads?](#)
 3. [Why Pthreads?](#)
 4. [Designing Threaded Programs](#)
2. [The Pthreads API](#)
3. [Thread Management](#)
 1. [Creating Threads](#)
 2. [Terminating Thread Execution](#)
 3. [Example: Pthread Creation and Termination](#)
 4. [Passing Arguments to Threads](#)
 5. [Thread Identifiers](#)
 6. [Joining Threads](#)
 7. [Detaching / Joining Threads](#)
 8. [Example: Joining Threads](#)
4. [Mutex Variables](#)
 1. [Mutex Variables Overview](#)
 2. [Creating / Destroying Mutexes](#)
 3. [Locking / Unlocking Mutexes](#)
 4. [Example: Using Mutexes](#)
5. [Condition Variables](#)
 1. [Condition Variables Overview](#)
 2. [Creating/Destroying Condition Variables](#)
 3. [Waiting / Signalling on Condition Variables](#)
 4. [Example: Using Condition Variables](#)
6. [LLNL Specific Information and Recommendations](#)
7. [Pthread Library Routines Reference](#)
8. [References and More Information](#)
9. [Exercise](#)
10. [Workshop Home](#)

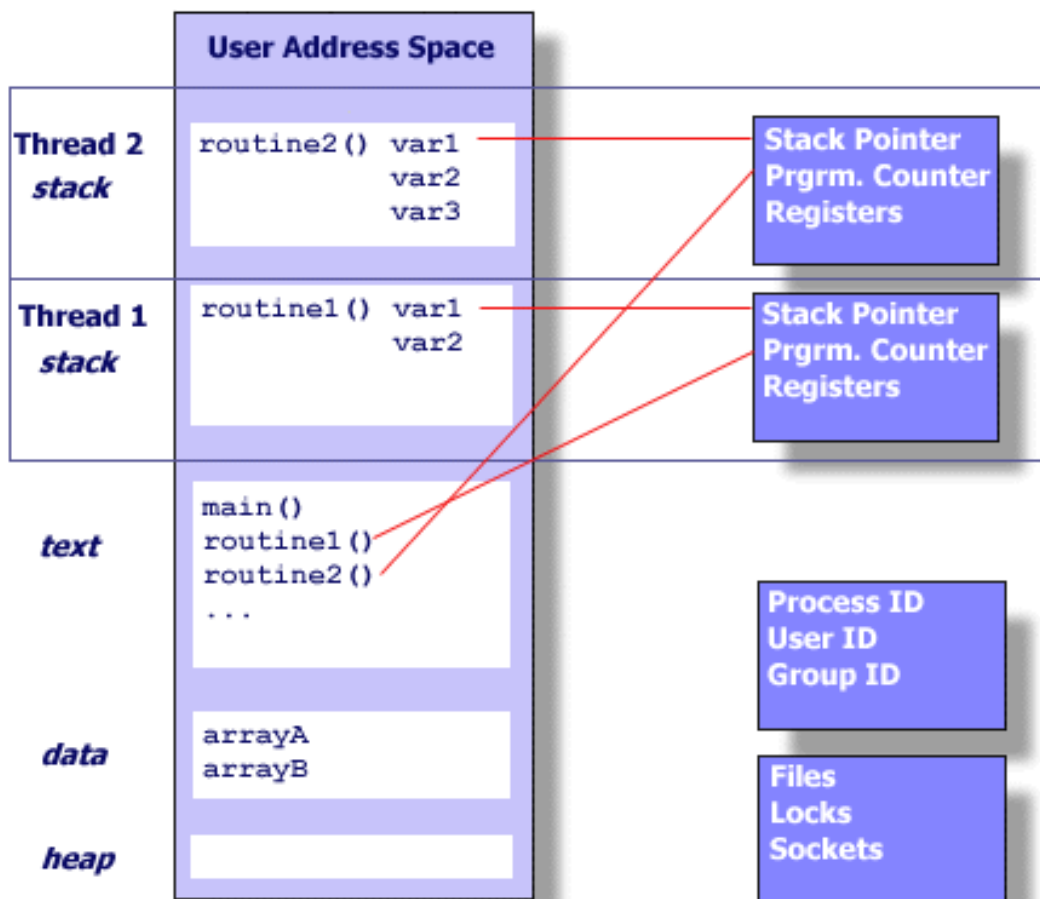
Pthreads Overview

What is a Thread?

- Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. But what does this mean?
- In the UNIX environment a thread:
 - Exists within a process and uses the process resources
 - Has its own independent flow of control as long as its parent process exists and the OS supports it
 - May share the process resources with other threads that act equally independently (and dependently)
 - Dies if the parent process dies - or something similar
- To the software developer, the concept of a "procedure" that runs independently from its main program may best describe a thread.
- Understanding what a thread means knowing the relationship between a process and a thread. A process is created by the operating system. Processes contain information about program resources and program execution state, including:
 - Process ID, process group ID, user ID, and group ID
 - Environment
 - Working directory.
 - Program instructions
 - Registers
 - Stack
 - Heap
 - File descriptors
 - Signal actions
 - Shared libraries
 - Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).



- Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities within a process.
- A thread can possess an independent flow of control and be schedulable because it maintains its own:
 - Stack pointer
 - Registers
 - Scheduling properties (such as policy or priority)
 - Set of pending and blocked signals
 - Thread specific data.



- A process can have multiple threads, all of which share the resources within a process and all of which execute within the same address space. Within a multi-threaded program, there are at any time multiple points of execution.
- Because threads within the same process share resources:
 - Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
 - Two pointers having the same value point to the same data.
 - Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

Pthreads Overview

What are Pthreads?

- Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or

Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's.

- Pthreads are defined as a set of C language programming types and procedure calls, implemented with a `pthread.h` header/include file and a thread library - though the this library may be part of another library, such as `libc`.
- There are several drafts of the POSIX threads standard. It is important to be aware of the draft number of a given implementation, because there are differences between drafts that can cause problems.

Pthreads Overview

Why Pthreads?

- The primary motivation for using Pthreads is to realize potential program performance gains.
- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

For example, the following table compares timing results for the **fork()** subroutine and the **pthread_create()** subroutine. Timings reflect 50,000 process/thread creations, were performed with the `time` utility, and units are in seconds.

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
IBM 332 MHz 604e 4 CPUs/node 512 MB Memory AIX 4.3	92.4	2.7	105.3	8.7	4.9	3.9
IBM 375 MHz POWER3 16 CPUs/node 16 GB Memory AIX 5.1	173.6	13.9	172.1	9.6	3.8	6.7
INTEL 2.2 GHz Xeon 2 CPU/node 2 GB Memory RedHat Linux 7.3	17.4	3.9	13.5	5.9	0.8	5.3

 [fork_vs_thread.txt](#)

- All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication.
- Threaded applications offer potential performance gains and practical advantages over non-threaded

applications in several other ways:

- Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
 - Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
 - Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.
- The primary motivation for considering the use of Pthreads on an SMP architecture is to achieve optimum performance. In particular, if an application is using MPI for on-node communications, there is a potential that performance could be greatly improved by using Pthreads for on-node data transfer instead.
 - For example: IBM's MPI provides the MP_SHARED_MEMORY environment variable to direct the use of shared memory for on-node MPI communications instead of switch communications. Without this, on-node MPI communications demonstrate serious performance degradation as the number of tasks increase.

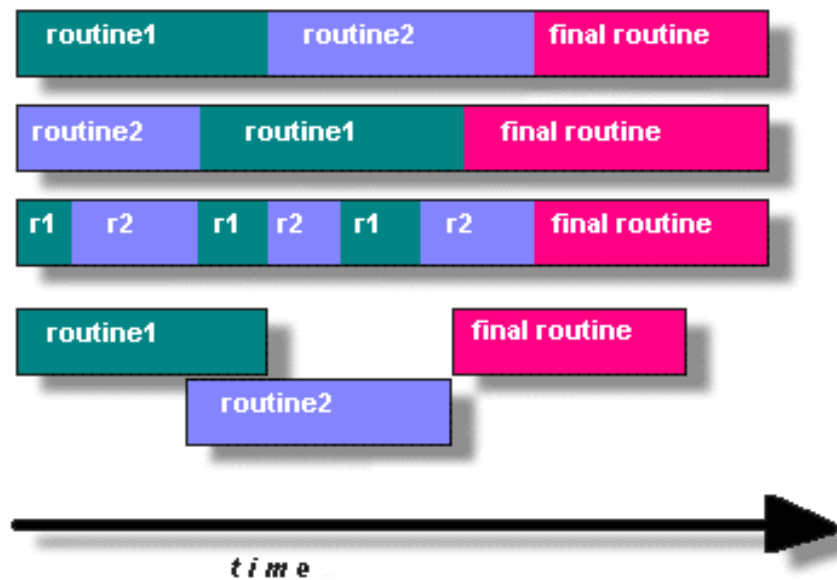
However, even with MP_SHARED_MEMORY set to "yes", on-node MPI communications can not compete with Pthreads:

- IBM 604e shared memory MPI bandwidth: @80 MB/sec per MPI task
- IBM POWER3 NH-2 shared memory MPI bandwidth: @275 MB/sec per MPI task
- Pthreads worst case: Every data reference by a thread requires a memory read. In this case, a thread's "bandwidth" is limited by the machine's memory-to-CPU bandwidth:
604e: 1.3 GB/sec
POWER3 NH-2: 16 GB/sec
- Pthreads best case: Data is local to a thread's cache offering much greater cache-CPU bandwidths.

Pthreads Overview

Designing Threaded Programs

- In order for a program to take advantage of Pthreads, it must be able to be organized into discrete, independent tasks which can execute concurrently. For example, if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.



- Tasks that may be suitable for threading include tasks that:
 - Block for potentially long waits
 - Use many CPU cycles
 - Must respond to asynchronous events
 - Are of lesser or greater importance than other tasks
 - Are able to be performed in parallel with other tasks
- Be careful if your application uses libraries or other objects that don't explicitly guarantee thread-safeness. When in doubt, assume that they are not thread-safe until proven otherwise.
- Several common models for threaded programs exist:
 - **Manager/worker:** a single thread, the *manager* assigns work to other threads, the *workers*. Typically, the manager handles all input and parcels out work to the other tasks. At least two forms of the manager/worker model are common: static worker pool and dynamic worker pool.
 - **Pipeline:** a task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread. An automobile assembly line best describes this model.
 - **Peer:** similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

The Pthreads API

- The Pthreads API is defined in the ANSI/IEEE POSIX 1003.1 - 1995 standard. Unlike MPI, this standard is not freely available on the Web - it must be [purchased from IEEE](#).
- The subroutines which comprise the Pthreads API can be informally grouped into three major classes:
 1. **Thread management:** The first class of functions work directly on threads - creating, detaching,

joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)

2. **Mutexes:** The second class of functions deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
3. **Condition variables:** The third class of functions address communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

- Naming conventions: All identifiers in the threads library begin with **pthread_**

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys

- The concept of opaque objects pervades the design of the API. The basic calls work to create or modify opaque objects - the opaque objects can be modified by calls to attribute functions, which deal with opaque attributes.
- The Pthreads API contains over 60 subroutines. This tutorial will focus on a subset of these - specifically, those which are most likely to be immediately useful to the beginning Pthreads programmer.
- The `pthread.h` header file must be included in each source file using the Pthreads library.
- The current POSIX standard is defined only for the C language. Fortran programmers can use wrappers around C function calls. Also, the IBM Fortran compiler provides a Pthreads API. See the XLF Language Reference, located with [IBM's Fortran documentation](#) for more information.
- A number of excellent books about Pthreads are available. Several of these are listed in the [References](#) section of this tutorial.

Thread Management

Creating Threads

- Initially, your `main()` program comprises a single, default thread. All other threads must be explicitly created by the programmer.

- Routines:

`pthread_create (thread, attr, start_routine, arg)`

- This routine creates a new thread and makes it executable. Typically, threads are first created from within `main()` inside a single process. Once created, threads are peers, and may create other threads.
- The `pthread_create` subroutine returns the new thread ID via the *thread* argument. The caller can use this thread ID to perform various operations on the thread. This ID should be checked to ensure that the thread was successfully created.
- The *attr* parameter is used to set thread attributes. You can specify a thread attributes object, or NULL for the default values. Thread attributes are discussed later.
- The *start_routine* is the C routine that the thread will execute once it is created.
- A single argument may be passed to *start_routine* via *arg*. It must be passed by reference as a pointer cast of type void.
- The maximum number of threads that may be created by a process is implementation dependent.



Question: After a thread has been created, how do you know when it will be scheduled to run by the operating system...especially on an SMP machine?

Answer

Thread Management

Terminating Thread Execution

- There are several ways in which a Pthread may be terminated:
 - The thread returns from its starting routine (the main routine for the initial thread).
 - The thread makes a call to the `pthread_exit` subroutine (covered below).
 - The thread is canceled by another thread via the `pthread_cancel` routine (not covered here).
 - The entire process is terminated due to a call to either the `exec` or `exit` subroutines.
- Routines:

`pthread_exit (status)`

- This routine is used to explicitly exit a thread. Typically, the `pthread_exit()` routine is called after a thread has completed its work and is no longer required to exist.
- If `main()` finishes before the threads it has created, and exits with `pthread_exit()`, the other threads will continue to execute. Otherwise, they will be automatically terminated when `main()` finishes.
- The programmer may optionally specify a termination *status*, which is stored as a void pointer for any thread that may join the calling thread.
- Cleanup: the `pthread_exit()` routine does not close files; any files opened inside the thread will remain open after the thread is terminated.
- Recommendation: Use `pthread_exit()` to exit from all threads...especially `main()`.

Thread Management

Example: Pthread Creation and Termination

- This simple example code creates 5 threads with the `pthread_create()` routine. Each thread prints a "Hello World!" message, and then terminates with a call to `pthread_exit()`.



Example Code - Pthread Creation and Termination

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS      5

void *PrintHello(void *threadid)
{
    printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t < NUM_THREADS; t++){
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

}

[Source](#) [Output](#)

Thread Management

Passing Arguments to Threads

- The `pthread_create()` routine permits the programmer to pass one argument to the thread start routine. For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the `pthread_create()` routine.
- All arguments must be passed by reference and cast to `(void *)`.



Question: How can you safely pass data to newly created threads, given their non-deterministic start-up and scheduling?

[Answer](#)

Example 1 - Thread Argument Passing

This code fragment demonstrates how to pass a simple integer to each thread. The calling thread uses a unique data structure for each thread, insuring that each thread's argument remains intact throughout the program.

```
int *taskids[NUM_THREADS];

for(t=0;t < NUM_THREADS;t++)
{
    taskids[t] = (int *) malloc(sizeof(int));
    *taskids[t] = t;
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) taskids[t]);
    ...
}
```

[Source](#) [Output](#)



Example 2 - Thread Argument Passing

This example shows how to setup/pass multiple arguments via a structure. Each thread receives a unique instance of the structure.

```

struct thread_data{
    int  thread_id;
    int  sum;
    char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
    struct thread_data *my_data;
    ...
    my_data = (struct thread_data *) threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    hello_msg = my_data->message;
    ...
}

int main (int argc, char *argv[])
{
    ...
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
    ...
}

```

[Source](#)[Output](#)

Example 3 - Thread Argument Passing (Incorrect)

This example performs argument passing incorrectly. The loop which creates threads modifies the contents of the address passed as an argument, possibly before the created threads can access it.

```

int rc, t;

for(t=0;t < NUM_THREADS;t++)
{
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &t);
}

```



```
...
}
```




Thread Management

Thread Identifiers

- Routines:

[`pthread\_self`](#) ()

[`pthread\_equal`](#) (`thread1`, `thread2`)

- The `pthread_self()` routine returns the unique, system assigned thread ID of the calling thread.
- The `pthread_equal()` routine compares two thread IDs. If the two IDs are different 0 is returned, otherwise a non-zero value is returned.
- Note that for both of these routines, the thread identifier objects are opaque and can not be easily inspected. Because thread IDs are opaque objects, the C language equivalence operator `==` should not be used to compare two thread IDs against each other, or to compare a single thread ID against another value.

Thread Management

Joining Threads

- "Joining" is one way to accomplish synchronization between threads. Two other ways, mutexes and condition variables will be discussed later.
- Routines:

[`pthread\_join`](#) (`threadid`, `status`)

- The `pthread_join()` subroutine blocks the calling thread until the specified *threadid* thread terminates.

- The programmer is able to obtain the target thread's termination return status if specified through `pthread_exit()`, in the *status* parameter.
- It is impossible to join a detached thread (discussed next)

Thread Management

Detaching / Joining Threads

- When a thread is created, one of its attributes defines whether it is joinable or detached. Detached means it can never be joined.
- Routines:

```
pthread_attr_init (attr)

pthread_attr_setdetachstate (attr, detachstate)

pthread_attr_getdetachstate (attr, detachstate)

pthread_attr_destroy (attr)

pthread_detach (threadid, status)
```

- To explicitly create a thread as joinable or detached, the `attr` argument in the `pthread_create()` routine is used. The typical 4 step process is:
 1. Declare a pthread attribute variable of the `pthread_attr_t` data type
 2. Initialize the attribute variable with `pthread_attr_init()`
 3. Set the attribute detached status with `pthread_attr_setdetachstate()`
 4. When done, free library resources used by the attribute with `pthread_attr_destroy()`
- The `pthread_detach()` routine can be used to explicitly detach a thread even though it was created as joinable. There is no converse routine.
- The final draft of the POSIX standard specifies that threads should be created as joinable. However, not all implementations may follow this.
- Recommendations:
 - If a thread requires joining, consider explicitly creating it as joinable. This provides portability as not all implementations may create threads as joinable by default.
 - If you know in advance that a thread will never need to join with another thread, consider creating it in a detached state. Some system resources may be able to be freed.

Thread Management

Example: Pthread Joining



Example Code - Pthread Joining

This example demonstrates how to "wait" for thread completions by using the Pthread join routine. Since not all current implementations of Pthreads create threads in a joinable state, the threads in this example are explicitly created in a joinable state so that they can be joined later.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS      3

void *BusyWork(void *null)
{
    int i;
    double result=0.0;
    for (i=0; i < 1000000; i++)
    {
        result = result + (double)random();
    }
    printf("result = %e\n",result);
    pthread_exit((void *) 0);
}

int main (int argc, char *argv[])
{
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int rc, t, status;

    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for(t=0;t < NUM_THREADS;t++)
    {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&thread[t], &attr, BusyWork, NULL);
        if (rc)
        {
            printf("ERROR; return code from pthread_create()
                    is %d\n", rc);
            exit(-1);
        }
    }

    /* Free attribute and wait for the other threads */
```

```
pthread_attr_destroy(&attr);
for(t=0;t < NUM_THREADS;t++)
{
    rc = pthread_join(thread[t], (void **)&status);
    if (rc)
    {
        printf("ERROR; return code from pthread_join()
               is %d\n", rc);
        exit(-1);
    }
    printf("Completed join with thread %d status= %d\n",t, status);
}

pthread_exit(NULL);
}
```



Mutex Variables

Overview

- Mutex is an abbreviation for "mutual exclusion". Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.
- A mutex variable acts like a "lock" protecting access to a shared data resource. The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful. No other thread can own that mutex until the owning thread unlocks that mutex. Threads must "take turns" accessing protected data.
- Mutexes can be used to prevent "race" conditions. An example of a race condition involving a bank transaction is shown below:

Thread 1	Thread 2	Balance
Read balance: \$1000		\$1000
	Read balance: \$1000	\$1000
	Deposit \$200	\$1000
Deposit \$200		\$1000
Update balance \$1000+\$200		\$1200
	Update balance \$1000+\$200	\$1200

- In the above example, a mutex should be used to lock the "Balance" while a thread is using this shared data resource.
- Very often the action performed by a thread owning a mutex is the updating of global variables. This is a safe way to ensure that when several threads update the same variable, the final value is the same as what it would be if only one thread performed the update. The variables being updated belong to a "critical section".
- A typical sequence in the use of a mutex is as follows:
 - Create and initialize a mutex variable
 - Several threads attempt to lock the mutex
 - Only one succeeds and that thread owns the mutex
 - The owner thread performs some set of actions
 - The owner unlocks the mutex
 - Another thread acquires the mutex and repeats the process
 - Finally the mutex is destroyed
- When several threads compete for a mutex, the losers block at that call - an unblocking call is available with "trylock" instead of the "lock" call.
- When protecting shared data, it is the programmer's responsibility to make sure every thread that needs to use a mutex does so. For example, if 4 threads are updating the same data, but only one uses a mutex, the data can still be corrupted.

Mutex Variables

Creating / Destroying Mutexes

- Routines:

```
pthread\_mutex\_init (mutex,attr)
```

```
pthread\_mutex\_destroy (mutex)
```

```
pthread\_mutexattr\_init (attr)
```

```
pthread\_mutexattr\_destroy (attr)
```

- Mutex variables must be declared with type `pthread_mutex_t`, and must be initialized before they can be used. There are two ways to initialize a mutex variable:
 1. Statically, when it is declared. For example:


```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

2. Dynamically, with the `pthread_mutex_init()` routine. This method permits setting mutex object attributes, *attr*.

The mutex is initially unlocked.

- The *attr* object is used to establish properties for the mutex object, and must be of type `pthread_mutexattr_t` if used (may be specified as `NULL` to accept defaults). The Pthreads standard defines three optional mutex attributes:
 - Protocol: Specifies the protocol used to prevent priority inversions for a mutex.
 - Prioceiling: Specifies the priority ceiling of a mutex.
 - Process-shared: Specifies the process sharing of a mutex.

Note that not all implementations may provide the three optional mutex attributes.

- The `pthread_mutexattr_init()` and `pthread_mutexattr_destroy()` routines are used to create and destroy mutex attribute objects respectively.
- `pthread_mutex_destroy()` should be used to free a mutex object which is no longer needed.

Mutex Variables

Locking / Unlocking Mutexes

- Routines:

[`pthread\_mutex\_lock`](#) (*mutex*)

[`pthread\_mutex\_trylock`](#) (*mutex*)

[`pthread\_mutex\_unlock`](#) (*mutex*)

- The `pthread_mutex_lock()` routine is used by a thread to acquire a lock on the specified *mutex* variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.
- `pthread_mutex_trylock()` will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately with a "busy" error code. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.
- `pthread_mutex_unlock()` will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data. An error will be returned if:
 - If the mutex was already unlocked
 - If the mutex is owned by another thread

- There is nothing "magical" about mutexes...in fact they are akin to a "gentlemen's agreement" between participating threads. It is up to the code writer to insure that the necessary threads all make the the mutex lock and unlock calls correctly. The following scenario demonstrates a logical error:

Thread 1	Thread 2	Thread 3
Lock	Lock	
$A = 2$	$A = A + 1$	$A = A * B$
Unlock	Unlock	



Question: When more than one thread is waiting for a locked mutex, which thread will be granted the lock first after it is released?

Answer

Mutex Variables

Example: Using Mutexes



Example Code - Using Mutexes

This example program illustrates the use of mutex variables in a threads program that performs a dot product. The main data is made available to all threads through a globally accessible structure. Each thread works on a different part of the data. The main thread waits for all the threads to complete their computations, and then it prints the resulting sum.

```
#include <pthread.h>
#include <stdio.h>
#include <malloc.h>
```

```
/*
```

```
The following structure contains the necessary information
to allow the function "dotprod" to access its input data and
place its output into the structure. This structure is
unchanged from the sequential version.
```

```
*/
```

```
typedef struct
```

```
{
```

```
    double    *a;
```

```
    double    *b;
```

```
    double    sum;
```

```
    int       vecLen;
```

```
} DOTDATA;
```

```
/* Define globally accessible variables and a mutex */
```

```
#define NUMTHRDS 4
```

```
#define VECLen 100
```

```
DOTDATA dotstr;
```

```

pthread_t callThd[NUMTHRDS];
pthread_mutex_t mutexsum;

/*
The function dotprod is activated when the thread is created.
All input to this routine is obtained from a structure
of type DOTDATA and all output from this function is written into
this structure. The benefit of this approach is apparent for the
multi-threaded program: when a thread is created we pass a single
argument to the activated function - typically this argument
is a thread number. All the other information required by the
function is accessed from the globally accessible structure.
*/

void *dotprod(void *arg)
{
    /* Define and use local variables for convenience */

    int i, start, end, offset, len ;
    double mysum, *x, *y;
    offset = (int)arg;

    len = dotstr.vecLen;
    start = offset*len;
    end   = start + len;
    x = dotstr.a;
    y = dotstr.b;

    /*
    Perform the dot product and assign result
    to the appropriate variable in the structure.
    */

    mysum = 0;
    for (i=start; i < end ; i++)
    {
        mysum += (x[i] * y[i]);
    }

    /*
    Lock a mutex prior to updating the value in the shared
    structure, and unlock it upon updating.
    */
    pthread_mutex_lock (&mutexsum);
    dotstr.sum += mysum;
    pthread_mutex_unlock (&mutexsum);

    pthread_exit((void*) 0);
}

/*
The main program creates threads which do all the work and then
print out result upon completion. Before creating the threads,
the input data is created. Since all threads update a shared structure,

```


we need a mutex for mutual exclusion. The main thread needs to wait for all threads to complete, it waits for each one of the threads. We specify a thread attribute value that allow the main thread to join with the threads it creates. Note also that we free up handles when they are no longer needed.

```

*/

int main (int argc, char *argv[])
{
    int i;
    double *a, *b;
    int status;
    pthread_attr_t attr;

    /* Assign storage and initialize values */
    a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
    b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

    for (i=0; i < VECLLEN*NUMTHRDS; i++)
    {
        a[i]=1;
        b[i]=a[i];
    }

    dotstr.vecLEN = VECLLEN;
    dotstr.a = a;
    dotstr.b = b;
    dotstr.sum=0;

    pthread_mutex_init(&mutexsum, NULL);

    /* Create threads to perform the dotproduct */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for(i=0;i < NUMTHRDS;i++)
    {
        /*
        Each thread works on a different set of data.
        The offset is specified by 'i'. The size of
        the data for each thread is indicated by VECLLEN.
        */
        pthread_create( &callThd[i], &attr, dotprod, (void *)i);
    }

    pthread_attr_destroy(&attr);

    /* Wait on the other threads */
    for(i=0;i < NUMTHRDS;i++)
    {
        pthread_join( callThd[i], (void **)&status);
    }

    /* After joining, print out the results and cleanup */
    printf ("Sum =  %f \n", dotstr.sum);
}

```

```

    free (a);
    free (b);
    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
}

```



Serial version



Pthreads version

Condition Variables

Overview

- Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.
- A condition variable is always used in conjunction with a mutex lock.
- A representative sequence for using condition variables is shown below.

Main Thread

- Declare and initialize global data/variables which require synchronization (such as "count")
- Declare and initialize a condition variable object
- Declare and initialize an associated mutex
- Create threads A and B to do work

Thread A

- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call `pthread_cond_wait()` to perform a blocking wait for signal from Thread-B. Note that a call to `pthread_cond_wait()` automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.

Thread B

- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon.
- Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
- Unlock mutex.
- Continue

- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue

Main Thread

Join / Continue

Condition Variables**Creating / Destroying Condition Variables**

- Routines:

[pthread_cond_init](#) (*condition*, *attr*)[pthread_cond_destroy](#) (*condition*)[pthread_condattr_init](#) (*attr*)[pthread_condattr_destroy](#) (*attr*)

- Condition variables must be declared with type `pthread_cond_t`, and must be initialized before they can be used. There are two ways to initialize a condition variable:
 1. Statically, when it is declared. For example:


```
pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
```
 2. Dynamically, with the `pthread_cond_init()` routine. The ID of the created condition variable is returned to the calling thread through the *condition* parameter. This method permits setting condition variable object attributes, *attr*.
 - The optional *attr* object is used to set condition variable attributes. There is only one attribute defined for condition variables: process-shared, which allows the condition variable to be seen by threads in other processes. The attribute object, if used, must be of type `pthread_condattr_t` (may be specified as NULL to accept defaults).
- Note that not all implementations may provide the process-shared attribute.
- The `pthread_condattr_init()` and `pthread_condattr_destroy()` routines are used to create and destroy condition variable attribute objects.
 - `pthread_cond_destroy()` should be used to free a condition variable that is no longer needed.

Condition Variables

Waiting / Signalling on Condition Variables

- Routines:

```
pthread_cond_wait (condition,mutex)
```

```
pthread_cond_signal (condition)
```

```
pthread_cond_broadcast (condition)
```

- `pthread_cond_wait()` blocks the calling thread until the specified *condition* is signalled. This routine should be called while *mutex* is locked, and it will automatically release the mutex while it waits. Should also unlock *mutex* after signal has been received.
- The `pthread_cond_signal()` routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after *mutex* is locked, and must unlock *mutex* in order for `pthread_cond_wait()` routine to complete.
- The `pthread_cond_broadcast()` routine should be used instead of `pthread_cond_signal()` if more than one thread is in a blocking wait state.
- It is a logical error to call `pthread_cond_signal()` before calling `pthread_cond_wait()`.



Proper locking and unlocking of the associated mutex variable is essential when using these routines. For example:

- Failing to lock the mutex before calling `pthread_cond_wait()` may cause it NOT to block.
- Failing to unlock the mutex after calling `pthread_cond_signal()` may not allow a matching `pthread_cond_wait()` routine to complete (it will remain blocked).

Condition Variables

Example: Using Condition Variables



Example Code - Using Condition Variables

This simple example code demonstrates the use of several Pthread condition variable routines. The main routine creates three threads. Two of the threads perform work and update a "count" variable. The third thread waits until the count variable reaches a specified value.

```
#include <pthread.h>;
#include <stdio.h>;

#define NUM_THREADS 3
#define TCOUNT 10
#define COUNT_LIMIT 12

int count = 0;
int thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *idp)
{
    int j,i;
    double result=0.0;
    int *my_id = idp;

    for (i=0; i < TCOUNT; i++) {
        pthread_mutex_lock(&count_mutex);
        count++;

        /*
        Check the value of count and signal waiting thread when condition is
        reached. Note that this occurs while mutex is locked.
        */
        if (count == COUNT_LIMIT) {
            pthread_cond_signal(&count_threshold_cv);
            printf("inc_count(): thread %d, count = %d Threshold reached.\n",
                *my_id, count);
        }
        printf("inc_count(): thread %d, count = %d, unlocking mutex\n",
            *my_id, count);
        pthread_mutex_unlock(&count_mutex);

        /* Do some work so threads can alternate on mutex lock */
        for (j=0; j < 1000; j++)
            result = result + (double)random();
    }
    pthread_exit(NULL);
}

void *watch_count(void *idp)
{
    int *my_id = idp;

    printf("Starting watch_count(): thread %d\n", *my_id);
```

```

/*
Lock mutex and wait for signal. Note that the pthread_cond_wait
routine will automatically and atomically unlock mutex while it waits.
Also, note that if COUNT_LIMIT is reached before this routine is run by
the waiting thread, the loop will be skipped to prevent pthread_cond_wait
from never returning.
*/
pthread_mutex_lock(&count_mutex);
while (count < COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %d Condition signal
        received.\n", *my_id);
}
pthread_mutex_unlock(&count_mutex);
pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    int i, rc;
    pthread_t threads[3];
    pthread_attr_t attr;

    /* Initialize mutex and condition variable objects */
    pthread_mutex_init(&count_mutex, NULL);
    pthread_cond_init (&count_threshold_cv, NULL);

    /*
    For portability, explicitly create threads in a joinable state
    so that they can be joined later.
    */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    pthread_create(&threads[0], &attr, inc_count, (void *)&thread_ids[0]);
    pthread_create(&threads[1], &attr, inc_count, (void *)&thread_ids[1]);
    pthread_create(&threads[2], &attr, watch_count, (void *)&thread_ids[2]);

    /* Wait for all threads to complete */
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

    /* Clean up and exit */
    pthread_attr_destroy(&attr);
    pthread_mutex_destroy(&count_mutex);
    pthread_cond_destroy(&count_threshold_cv);
    pthread_exit(NULL);
}

```



LLNL Specific Information and Recommendations

This section describes details specific to Livermore Computing's systems.

Implementations:

- All LC production systems include a Pthreads implementation that follows draft 10 (final) of the POSIX standard. This is the preferred implementation.
- For compatibility with earlier implementations, the IBM compilers provide a means to use the draft 7 version of Pthreads and the Compaq Tru64 compilers a means to use the draft 4 version.
- Implementation Notes:
 - The three mutex attributes, protocol, prioceiling and process-shared are optional and may not be supported on all platforms.
 - The process-shared condition variable attribute is optional and may not supported on all platforms.
 - Implementations differ in the maximum number of threads that a process may create.

Compiling:

- LC maintains a number of compilers, and usually several different versions of each - see the [LC's Supported Compilers](#) web page.
- The more commonly used compile commands for pthreads codes are listed in the table below.

Platform	Compiler Command	Description
IBM¹ AIX	xlc_r / cc_r	C (ANSI / non-ANSI)
	xlc_r	C++
	xlf_r -qnosave xlf90_r -qnosave	Fortran - using IBM's Pthreads API (non-portable)
INTEL LINUX	icc -pthread	C / C++
COMPAQ Tru64	cc -pthread	C
	cxx -pthread	C++
All Platforms	gcc -pthread	GNU C
	g++ -pthread	GNU C++
	guidec -pthread	KAI C
	KCC -pthread	KAI C++

¹ All LC IBM compilers are aliased to their thread-safe command. For example, xlc really uses xlc_r.

Mixing MPI with Pthreads:

- Programs that contain both MPI and Pthreads are common and easy to develop on all LC systems.
- Design:
 - Each MPI process typically creates and then manages N threads, where N makes the best use of the available CPUs/node.
 - Finding the best value for N will vary with the platform and your application's characteristics.
 - For ASCI White systems with two communication adapters per node, it may prove more efficient to use two (or more) MPI tasks per node.
 - In general, there may be problems if multiple threads make MPI calls. The program may fail or behave unexpectedly. If MPI calls must be made from within a thread, they should be made only by one thread.
- Compiling:
 - Use the appropriate MPI compile command for the platform and language of choice
 - Be sure to include the required flag as in the table above (-pthread or -qnosave)
 - MPICH is not thread safe
- An example code that uses both MPI and Pthreads is available below. The serial, threads-only, MPI-only and MPI-with-threads versions demonstrate one possible progression.
 - [Serial](#)
 - [Pthreads only](#)
 - [MPI only](#)
 - [MPI with pthreads](#)
 - [makefile](#) (for IBM SP)

Pthread Library Routines Reference

Pthread Functions	
Thread Management	pthread_create
	pthread_exit
	pthread_join
	pthread_once
	pthread_kill
	pthread_self

	pthread_equal
	pthread_yield
	pthread_detach
Thread Specific Data	pthread_key_create
	pthread_key_delete
	pthread_getspecific
	pthread_setspecific
Thread Cancellation	pthread_cancel
	pthread_cleanup_pop
	pthread_cleanup_push
	pthread_setcancelstate
	pthread_getcancelstate
	pthread_testcancel
Thread Scheduling	pthread_getschedparam
	pthread_setschedparam
Signals	pthread_sigmask
Pthread Attribute Functions	
Basic Management	pthread_attr_init
	pthread_attr_destroy
Detachable or Joinable	pthread_attr_setdetachstate
	pthread_attr_getdetachstate
Specifying Stack Information	pthread_attr_getstackaddr
	pthread_attr_getstacksize
	pthread_attr_setstackaddr
	pthread_attr_setstacksize
Thread Scheduling Attributes	pthread_attr_getschedparam
	pthread_attr_setschedparam
	pthread_attr_getschedpolicy

	pthread_attr_setschedpolicy
	pthread_attr_setinheritsched
	pthread_attr_getinheritsched
	pthread_attr_setscope
	pthread_attr_getscope
Mutex Functions	
Mutex Management	pthread_mutex_init
	pthread_mutex_destroy
	pthread_mutex_lock
	pthread_mutex_unlock
	pthread_mutex_trylock
Priority Management	pthread_mutex_setprioceiling
	pthread_mutex_getprioceiling
Mutex Attribute Functions	
Basic Management	pthread_mutexattr_init
	pthread_mutexattr_destroy
Sharing	pthread_mutexattr_getpshared
	pthread_mutexattr_setpshared
Protocol Attributes	pthread_mutexattr_getprotocol
	pthread_mutexattr_setprotocol
Priority Management	pthread_mutexattr_setprioceiling
	pthread_mutexattr_getprioceiling
Condition Variable Functions	
Basic Management	pthread_cond_init
	pthread_cond_destroy
	pthread_cond_signal
	pthread_cond_broadcast
	pthread_cond_wait

	pthread_cond_timedwait
Condition Variable Attribute Functions	
Basic Management	pthread_condattr_init
	pthread_condattr_destroy
Sharing	pthread_condattr_getpshared
	pthread_condattr_setpshared

References and More Information

- "Pthreads Programming". B. Nichols et al. O'Reilly and Associates.
- "Threads Primer". B. Lewis and D. Berg. Prentice Hall
- "Programming With POSIX Threads". D. Butenhof. Addison Wesley
www.awl.com/cseng/titles/0-201-63392-2
- "Programming With Threads". S. Kleiman et al. Prentice Hall
- IBM Fortran Compiler Documentation
www-4.ibm.com/software/ad/fortran
- IBM C/C++ Compiler Documentation
www-4.ibm.com/software/ad/caix

Last Modified: blaiseb@llnl.gov

UCRL-MI-133316

YoLinux Tutorial: POSIX thread (pthread) libraries

The POSIX thread libraries are a standards based thread API for C/C++. It allows one to spawn a new concurrent process flow. It is most effective on multiprocessor systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing. Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process. While most effective on a multiprocessor system, gains are also found on uniprocessor systems which exploit latency in I/O and other system functions which may halt process execution. (One thread may execute while another is waiting for I/O or some other system latency.) Parallel programming technologies such as MPI and PVM are used in a distributed computing environment while threads are limited to a single computer system. All threads within a process share the same address space. A thread is spawned by defining a function and it's arguments which will be processed in the thread. The purpose of using the POSIX thread library in your software is to execute software faster.

Contents:

- [Thread Basics](#)
 - [Thread Creation and Termination](#)
 - [Thread Synchronization](#)
 - [Thread Scheduling](#)
 - [Thread Pitfalls](#)
 - [Thread Debugging](#)
 - [Thread Man Pages](#)
 - [Links](#)
 - [Books](#)
-

YoLinux Programming Tutorials:

- [C++ on Linux](#)
- [C++ STL \(Standard Template Library\) example of a linked list using a list](#)
- [C++ string class examples](#)
- [X-emacs and C++ development](#)
- [C++ Structure Example and Tutorial](#)
- [Linux software development tutorial](#)



Thread Basics:

- Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.
- Threads in the same process share:
 - Process instructions
 - Most data
 - open files (descriptors)
 - signals and signal handlers
 - current working directory
 - User and group id
- Each thread has a unique:

- Thread ID
- set of registers, stack pointer
- stack for local variables, return addresses
- signal mask
- priority
- Return value: errno
- pthread functions return "0" if ok.

Thread Creation and Termination:

Example: pthread1.c

```
#include <stdio.h>
#include <pthread.h>

void print_message_function( void *ptr );

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  iret1, iret2;

    /* Create independant threads each of which will execute function */

    iret1 = pthread\_create( &thread1, NULL, (void*)&print_message_function, (void*)
message1);
    iret2 = pthread_create( &thread2, NULL, (void*)&print_message_function, (void*)
message2);

    /* Wait till threads are complete before main continues. Unless we */
    /* wait we run the risk of executing an exit which will terminate */
    /* the process and all threads before the threads have completed. */

    pthread\_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}

void print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

Compile: cc -lpthread pthread1.c

Run: ./a.out

Results:

```
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

Details:

- In this example the same function is used in each thread. The arguments are different. The functions need not be the same.
- Threads terminate by explicitly calling `pthread_exit`, by letting the function return, or by a call to the function `exit` which will terminate the process including any threads.
- Function call: [`pthread\_create`](#)

```
int pthread_create(pthread_t * thread,
                  const pthread_attr_t * attr,
                  void * (*start_routine)(void
*),
                  void *arg);
```

Arguments:

- `thread` - returns the thread id. (unsigned long int defined in `bits/pthreadtypes.h`)
 - `attr` - Set to NULL if default thread attributes are used. (else define members of the struct `pthread_attr_t` defined in `bits/pthreadtypes.h`) Attributes include:
 - detached state (joinable? Default: `PTHREAD_CREATE_JOINABLE`. Other option: `PTHREAD_CREATE_DETACHED`)
 - scheduling policy (real-time? `PTHREAD_INHERIT_SCHED`, `PTHREAD_EXPLICIT_SCHED`, `SCHED_OTHER`)
 - scheduling parameter
 - inheritsched attribute (Default: `PTHREAD_EXPLICIT_SCHED` Inherit from parent thread: `PTHREAD_INHERIT_SCHED`)
 - scope (Kernel threads: `PTHREAD_SCOPE_SYSTEM` User threads: `PTHREAD_SCOPE_PROCESS` Pick one or the other not both.)
 - guard size
 - stack address (See `unistd.h` and `bits/posix_opt.h` `_POSIX_THREAD_ATTR_STACKADDR`)
 - stack size (default minimum `PTHREAD_STACK_SIZE` set in `pthread.h`),
 - `void * (*start_routine)` - pointer to the function to be threaded. Function has a single argument: pointer to void.
 - `*arg` - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.
- Function call: [`pthread\_exit`](#)

```
void pthread_exit(void *retval);
```

Arguments:

- `retval` - Return value of thread.

This routine kills the thread. The `pthread_exit` function never returns. If the thread is not detached, the thread id and return value may be examined from another thread by using `pthread_join`.

Note: the return pointer `*retval`, must not be of local scope otherwise it would cease to exist once the thread terminates.

Thread Synchronization:

The threads library provides three synchronization mechanisms:

- mutexes - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.
- joins - Make a thread wait till others are complete (terminated).
- condition variables - data type `pthread_cond_t`

Mutexes:

Mutexes are used to prevent data inconsistencies due to race conditions. A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed. Mutexes are used for serializing shared resources. Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it. One can apply a mutex to protect a segment of memory ("critical region") from other threads.

Example threaded function:

Without Mutex	With Mutex		
<pre>int counter=0; /* Function C */ void functionC() { counter++ }</pre>	<pre>/* Note scope of variable and mutex are the same */ pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER; int counter=0; /* Function C */ void functionC() { pthread_mutex_lock(&mutex1); counter++ pthread_mutex_unlock(&mutex1); }</pre>		
Possible execution sequence			
Thread 1	Thread 2	Thread 1	Thread 2
counter = 0	counter = 0	counter = 0	counter = 0
counter = 1	counter = 1	counter = 1	Thread 2 locked out. Thread 1 has exclusive use of variable counter
			counter = 2

If register load and store operations for the incrementing of variable counter occurs with unfortunate timing, it is theoretically possible to have each thread increment and overwrite the same variable with the same value. Another possibility is that thread two would first increment counter locking out thread one until complete and then thread one would increment it to 2.

Sequence	Thread 1	Thread 2
1	counter = 0	counter=0
2	Thread 1 locked out. Thread 2 has exclusive use of variable counter	counter = 1
3	counter = 2	

Code listing: mutex1.c

```
#include <stdio.h>
#include <pthread.h>

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

main()
{
    int rc1, rc2;
    pthread_t thread1, thread2;
```

```

/* Create independant threads each of which will execute functionC */

if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
{
    printf("Thread creation failed: %d\n", rc1);
}

if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
{
    printf("Thread creation failed: %d\n", rc2);
}

/* Wait till threads are complete before main continues. Unless we */
/* wait we run the risk of executing an exit which will terminate */
/* the process and all threads before the threads have completed. */

pthread_join( thread1, NULL);
pthread_join( thread2, NULL);

exit(0);
}

void *functionC()
{
    pthread_mutex_lock( &mutex1 );
    counter++;
    printf("Counter value: %d\n",counter);
    pthread_mutex_unlock( &mutex1 );
}

```

Compile: cc -lpthread mutex1.c

Run: ./a.out

Results:

```

Counter value: 1
Counter value: 2

```

When a mutex lock is attempted against a mutex which is held by another thread, the thread is blocked until the mutex is unlocked. When a thread terminates, the mutex does not unless explicitly unlocked. Nothing happens by default.

Joins:

A join is performed when one wants to wait for a thread to finish. A thread calling routine may launch multiple threads then wait for them to finish to get the results. One wait for the completion of the threads with a join.

Sample code: join1.c


```

#include <stdio.h>
#include <pthread.h>

#define NTHREADS 10
void *thread_function();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

main()
{
    pthread_t thread_id[NTHREADS];
    int i, j;

    for(i=0; i < NTHREADS; i++)
    {
        pthread\_create( &thread_id[i], NULL, &thread_function, NULL );
    }

    for(j=0; j < NTHREADS; j++)
    {
        pthread\_join( thread_id[j], NULL);
    }

    /* Now that all threads are complete I can print the final result.      */
    /* Without the join I could be printing a value before all the threads */
    /* have been completed.                                                */

    printf("Final counter value: %d\n", counter);
}

void *thread_function()
{
    printf("Thread number %ld\n", pthread\_self());
    pthread_mutex_lock( &mutex1 );
    counter++;
    pthread_mutex_unlock( &mutex1 );
}

```

Compile: cc -lpthread join1.c

Run: ./a.out

Results:

```

Thread number 1026
Thread number 2051
Thread number 3076
Thread number 4101
Thread number 5126
Thread number 6151
Thread number 7176
Thread number 8201
Thread number 9226
Thread number 10251
Final counter value: 10

```

Condition Variables:

A condition variable is a variable of type `pthread_cond_t` and is used with the appropriate functions for waiting and later, process continuation. The condition variable mechanism allows threads to suspend execution and relinquish the

processor until some condition is true. A condition variable must always be associated with a mutex to avoid a race condition created by one thread preparing to wait and another thread which may signal the condition before the first thread actually waits on it resulting in a deadlock. The thread will be perpetually waiting for a signal that is never sent. Any mutex can be used, there is no explicit link between the mutex and the condition variable.

Functions used in conjunction with the condition variable:

- Creating/Destroying:
 - [pthread_cond_init](#)
 - `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
 - [pthread_cond_destroy](#)
- Waiting on condition:
 - [pthread_cond_wait](#)
 - [pthread_cond_timedwait](#) - place limit on how long it will block.
- Waking thread based on condition:
 - [pthread_cond_signal](#)
 - [pthread_cond_broadcast](#) - wake up all threads blocked by the specified condition variable.

Example code: cond1.c

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t count_mutex      = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_cond  = PTHREAD_COND_INITIALIZER;

void *functionCount1();
void *functionCount2();
int  count = 0;
#define COUNT_DONE  10
#define COUNT_HALT1 3
#define COUNT_HALT2 6

main()
{
    pthread_t thread1, thread2;

    pthread_create( &thread1, NULL, &functionCount1, NULL);
    pthread_create( &thread2, NULL, &functionCount2, NULL);
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    exit(0);
}

void *functionCount1()
{
    for(;;)
    {
        pthread\_mutex\_lock( &condition_mutex );
        while( count >= COUNT_HALT1 && count <= COUNT_HALT2 )
        {
            pthread\_cond\_wait( &condition_cond, &condition_mutex );
        }
        pthread\_mutex\_unlock( &condition_mutex );

        pthread_mutex_lock( &count_mutex );
        count++;
    }
}
```

```

        printf("Counter value functionCount1: %d\n",count);
        pthread_mutex_unlock( &count_mutex );

        if(count >= COUNT_DONE) return(NULL);
    }
}

void *functionCount2()
{
    for(;;)
    {
        pthread_mutex_lock( &condition_mutex );
        if( count < COUNT_HALT1 || count > COUNT_HALT2 )
        {
            pthread\_cond\_signal( &condition_cond );
        }
        pthread_mutex_unlock( &condition_mutex );

        pthread_mutex_lock( &count_mutex );
        count++;
        printf("Counter value functionCount2: %d\n",count);
        pthread_mutex_unlock( &count_mutex );

        if(count >= COUNT_DONE) return(NULL);
    }
}

```

Compile: `cc -lpthread cond1.c`

Run: `./a.out`

Results:

```

Counter value functionCount1: 1
Counter value functionCount1: 2
Counter value functionCount1: 3
Counter value functionCount2: 4
Counter value functionCount2: 5
Counter value functionCount2: 6
Counter value functionCount2: 7
Counter value functionCount1: 8
Counter value functionCount1: 9
Counter value functionCount1: 10
Counter value functionCount2: 11

```

Note that `functionCount1()` was halted while count was between the values `COUNT_HALT1` and `COUNT_HALT2`. The only thing that has been ensured is that `functionCount2` will increment the count between the values `COUNT_HALT1` and `COUNT_HALT2`. Everything else is random.

The logic conditions (the "if" and "while" statements) must be chosen to insure that the "signal" is executed if the "wait" is ever processed. Poor software logic can also lead to a deadlock condition.

Note: Race conditions abound with this example because count is used as the condition and can't be locked in the while statement without causing deadlock. I'll work on a cleaner example but it is an example of a condition variable.

Thread Scheduling:

When this option is enabled, each thread may have its own scheduling properties. Scheduling attributes may be specified:

- during thread creation

- by dynamically by changing the attributes of a thread already created
- by defining the effect of a mutex on the thread's scheduling when creating a mutex
- by dynamically changing the scheduling of a thread during synchronization operations.

The threads library provides default values that are sufficient for most cases.

Thread Pitfalls:

- **Race conditions:** While the code may appear on the screen in the order you wish the code to execute, threads are scheduled by the operating system and are executed at random. It cannot be assumed that threads are executed in the order they are created. They may also execute at different speeds. When threads are executing (racing to complete) they may give unexpected results (race condition). Mutexes and joins must be utilized to achieve a predictable execution order and outcome.
- **Thread safe code:** The threaded routines must call functions which are "thread safe". This means that there are no static or global variables which other threads may clobber or read assuming single threaded operation. If static or global variables are used then mutexes must be applied or the functions must be re-written to avoid the use of these variables. In C, local variables are dynamically allocated on the stack. Therefore, any function that does not use static data or other shared resources is thread-safe. Thread-unsafe functions may be used by only one thread at a time in a program and the uniqueness of the thread must be ensured. Many non-reentrant functions return a pointer to static data. This can be avoided by returning dynamically allocated data or using caller-provided storage. An example of a non-thread safe function is `strtok` which is also not re-entrant. The "thread safe" version is the re-entrant version `strtok_r`.
- **Mutex Deadlock:** This condition occurs when a mutex is applied but then not "unlocked". This causes program execution to halt indefinitely. It can also be caused by poor application of mutexes or joins. Be careful when applying two or more mutexes to a section of code. If the first `pthread_mutex_lock` is applied and the second `pthread_mutex_lock` fails due to another thread applying a mutex, the first mutex may eventually lock all other threads from accessing data including the thread which holds the second mutex. The threads may wait indefinitely for the resource to become free causing a deadlock. It is best to test and if failure occurs, free the resources and stall before retrying.

```
...
pthread_mutex_lock(&mutex_1);
while ( pthread_mutex_trylock(&mutex_2) ) /* Test if already locked */
{
    pthread_mutex_unlock(&mutex_1); /* Free resource to avoid deadlock */
    ...
    /* stall here */
    ...
    pthread_mutex_lock(&mutex_1);
}
count++;
pthread_mutex_unlock(&mutex_1);
pthread_mutex_unlock(&mutex_2);
...
```

The order of applying the mutex is also important. The following code segment illustrates a potential for deadlock:

```

void *function1()
{
    ...
    pthread_mutex_lock(&lock1);           - Execution step 1
    pthread_mutex_lock(&lock2);           - Execution step 3 DEADLOCK!!!
    ...
    pthread_mutex_lock(&lock2);
    pthread_mutex_lock(&lock1);
    ...
}

void *function2()
{
    ...
    pthread_mutex_lock(&lock2);           - Execution step 2
    pthread_mutex_lock(&lock1);
    ...
    pthread_mutex_lock(&lock1);
    pthread_mutex_lock(&lock2);
    ...
}

main()
{
    ...
    pthread_create(&thread1, NULL, function1, NULL);
    pthread_create(&thread2, NULL, function1, NULL);
    ...
}

```

If function1 aquires the first mutex and function2 aquires the second, all resources are tied up and locked.

- Condition Variable Deadlock: The logic conditions (the "if" and "while" statements) must be chosen to insure that the "signal" is executed if the "wait" is ever processed.

Thread Debugging:

- [Thread Debug Interface \(TDI\) and Thread Debugging Commands of the GDB](#)
- [Smart GDB](#)

Thread Man Pages:

- [pthread_atfork](#) - register handlers to be called at fork(2) time
- [pthread_attr_destroy](#) [pthread_attr_init] - thread creation attributes
- [pthread_attr_getdetachstate](#) [pthread_attr_init] - thread creation attributes
- [pthread_attr_getinheritsched](#) [pthread_attr_init] - thread creation attributes
- [pthread_attr_getschedparam](#) [pthread_attr_init] - thread creation attributes
- [pthread_attr_getschedpolicy](#) [pthread_attr_init] - thread creation attributes
- [pthread_attr_getscope](#) [pthread_attr_init] - thread creation attributes
- [pthread_attr_init](#) - thread creation attributes
- [pthread_attr_setdetachstate](#) [pthread_attr_init] - thread creation attributes
- [pthread_attr_setinheritsched](#) [pthread_attr_init] - thread creation attributes
- [pthread_attr_setschedparam](#) [pthread_attr_init] - thread creation attributes
- [pthread_attr_setschedpolicy](#) [pthread_attr_init] - thread creation attributes
- [pthread_attr_setscope](#) [pthread_attr_init] - thread creation attributes
- [pthread_cancel](#) - thread cancellation
- [pthread_cleanup_pop](#) [pthread_cleanup_push] - install and remove cleanup handlers

- [pthread_cleanup_pop_restore_np](#) [pthread_cleanup_push] - install and remove cleanup handlers
- [pthread_cleanup_push](#) - install and remove cleanup handlers
- [pthread_cleanup_push_defer_np](#) [pthread_cleanup_push] - install and remove cleanup handlers
- [pthread_condattr_destroy](#) [pthread_condattr_init] - condition creation attributes
- [pthread_condattr_init](#) - condition creation attributes
- [pthread_cond_broadcast](#) [pthread_cond_init] - operations on conditions
- [pthread_cond_destroy](#) [pthread_cond_init] - operations on conditions
- [pthread_cond_init](#) - operations on conditions
- [pthread_cond_signal](#) [pthread_cond_init] - operations on conditions
- [pthread_cond_timedwait](#) [pthread_cond_init] - operations on conditions
- [pthread_cond_wait](#) [pthread_cond_init] - operations on conditions
- [pthread_create](#) - create a new thread
- [pthread_detach](#) - put a running thread in the detached state
- [pthread_equal](#) - compare two thread identifiers
- [pthread_exit](#) - terminate the calling thread
- [pthread_getschedparam](#) [pthread_setschedparam] - control thread scheduling parameters
- [pthread_getspecific](#) [pthread_key_create] - management of thread-specific data
- [pthread_join](#) - wait for termination of another thread
- [pthread_key_create](#) - management of thread-specific data
- [pthread_key_delete](#) [pthread_key_create] - management of thread-specific data
- [pthread_kill_other_threads_np](#) - terminate all threads in program except calling thread
- [pthread_kill](#) [pthread_sigmask] - handling of signals in threads
- [pthread_mutexattr_destroy](#) [pthread_mutexattr_init] - mutex creation attributes
- [pthread_mutexattr_getkind_np](#) [pthread_mutexattr_init] - mutex creation attributes
- [pthread_mutexattr_init](#) - mutex creation attributes
- [pthread_mutexattr_setkind_np](#) [pthread_mutexattr_init] - mutex creation attributes
- [pthread_mutex_destroy](#) [pthread_mutex_init] - operations on mutexes
- [pthread_mutex_init](#) - operations on mutexes
- [pthread_mutex_lock](#) [pthread_mutex_init] - operations on mutexes
- [pthread_mutex_trylock](#) [pthread_mutex_init] - operations on mutexes
- [pthread_mutex_unlock](#) [pthread_mutex_init] - operations on mutexes
- [pthread_once](#) - once-only initialization
- [pthread_self](#) - return identifier of current thread
- [pthread_setcancelstate](#) [pthread_cancel] - thread cancellation
- [pthread_setcanceltype](#) [pthread_cancel] - thread cancellation
- [pthread_setschedparam](#) - control thread scheduling parameters
- [pthread_setspecific](#) [pthread_key_create] - management of thread-specific data
- [pthread_sigmask](#) - handling of signals in threads
- [pthread_testcancel](#) [pthread_cancel] - thread cancellation

Links:

- [Next Generation POSIX Threading](#) - IBM Developerworks
- [Native Posix Thread Library for Linux](#)
- [Introduction to Programming Threads](#)
- [LinuxThreads Frequently Asked Questions](#)
- [Fundamentals Of Multithreading](#) - Paul Mazzucco
- [Getting Started With POSIX Threads](#)
- [ITS: Introduction to Threads](#)
- [GNU Portable Threads](#)
- [Introduction of threads for Solaris, Linux, and Windows](#)
- [Comparison of thread implementations](#)
- [comp.programming.threads FAQ](#)
- [An in-depth description of PMPthread internal queue functions.](#)

- [Examples](#)
- [Pthreads tutorial and examples of thread problems](#) - by Andrae Muys
- [Valgrind KDE thread checker: Helgrind](#)
- [Sun's Multithreaded Programming Guide](#) - Not Linux but a good reference.
- [IBM pthread documentationFSU Pthreads \(POSIX Threads\)](#)
- [Linux-mag.com: Concurrent Programming Topics](#) - semaphores, condition variables
- [Linux-mag.com: The Fibers of Threads](#) - Discussion of how Linux threads work
- [Linux Expo POSIX threads Lecture](#)
- Platform independent threads:
 - [Gnome GLib 2.0 threads](#) - Thread abstraction; including mutexes, conditions and thread private data. [\[example\]](#)
 - [OmniORB \(CORBA\) Thread Library](#)
 - [zThreads](#)
- **C++ Thread classes:**
 - [GNU: Common C++](#) - support for threading, sockets, file access, daemons, persistence, serial I/O, XML parsing and system services
 - [ACE: Adaptive Communication Environment](#) - C++ interface
 - ACE programmers guide: [\[pdf\]](#) ([see page 29 for threads](#))
 - [Thread management examples using ACE](#)
 - [Hood](#) - A C++ Threads Library for Multiprogrammed Multiprocessors
 - [C++ Thread classes](#) - sourceforge
 - [QpThread](#)

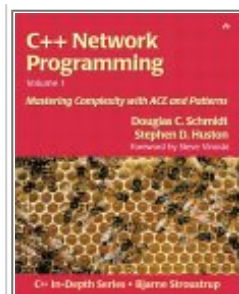
News Groups:

- comp.programming.threads
- comp.unix.solaris



Books:

	Pthreads Programming A POSIX Standard for Better Multiprocessing By Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farrell ISBN #1-56592-115-1, O'Reilly	
	Programming with POSIX(R) Threads By David R. Butenhof ISBN #0201633922, Addison Wesley Pub. Co.	



C++ Network Programming Volume 1
By Douglas C. Schmidt, Stephen D. Huston
ISBN #0201604647, Addison Wesley Pub. Co.

Covers ACE (ADAPTIVE Communication Environment) open-source framework view of threads and other topics.



See <http://YoLinux.com> for more Linux information and tutorials

Return to [YoLinux Tutorial Index](#)

Copyright © 2002, 2003, 2004 by *Greg Ippolito*



▲ Vorlesungen

▶ Gliederung

▶ Netzressourcen zu in der Vorlesung behandelten Themen

▼ Script

▼ Literatur

▶ Alte Klausuren

▲ Inhalt

Die Veranstaltung widmet sich, nach einer allgemein gehaltenen Einführung in den Themenkomplex Thread-basierter paralleler Verarbeitung, den speziellen Möglichkeiten der Umsetzung mit der Java-Plattform. Zunächst wird der Thread-Unterstützung durch die plattformunabhängige Java-API (in der Version 1.4) sowie der virtuellen Maschine Raum gewidmet. Aufbauend darauf werden mit Synchronisation und Kommunikation die zentralen Themen der Parallelverarbeitung diskutiert. Zur Veranschaulichung werden klassische Probleme -- wie die *hungrigen Philosophen*, welche auch aus verwandten Gebieten bekannt sind, -- herangezogen. Darüberhinaus werden den spezifischen Hochsprachen-Charakteristika in Java, etwa durch in der Syntax vorgesehene Schlüsselworte zur Kontrolle und Beeinflussung der Ausführung oder das Prioritätskonzept, eingeführt und an Beispielen vorgestellt. Den Abschluß bildet eine ausführliche Betrachtung der Anwendbarkeit der vorgestellten Konzepte und Sprachmittel zur Lösung verschiedener Probleme. Hierbei wird neben grundsätzlichen algorithmischen Aspekten auch die Verwendung von Threads in graphischen Oberflächen und der Java-API in den Blickpunkt gerückt.

▲ Scriptum zur Vorlesung

[HTML online Version](#)

[PDF-Version](#)

[XML Version](#)

[Tief verlinkte PDF-Version](#)

▲ Literatur

Empfohlene Primärliteratur

... sollte unbedingt gelesen werden!

Rainer Oechsle

[Parallele Programmierung mit Java Threads](#)

Das Buch ist auch in einigen Exemplaren in der Bibliothek vorhanden!

Scott Oaks, Henry Wong

Java Threads [1. Auflage](#), [2. Auflage](#)

JavaSoft (eds.)

[Java Tutorial](#)

Vertiefendes

... bietet zusätzliche Hintergründe und Einblicke, sollte bei Bedarf gelesen werden!

Jeff Magee, Jeff Kramer

[Concurrency: State Models & Java Programs](#)

Doug Lea

[Concurrent Programming in Java: Design Principles and Patterns](#)
[Web-Seite mit Ergänzungen und Korrekturen zum Buch](#)

Weiterführendes

... kann bei Zeit und Gelegenheit gelesen werden!

Paul Hyde

[Java Thread Programming](#)

Sonstige Literatur zum Thema

Open UNIX/POSIX Spezifikation
Standardisierte Threads unter UNIX

[HTML](#)

Heinz Kredel, Akitoshi Yoshida

[Thread- und Netzwerk-Programmierung mit Java](#)

Brett Spell

[Professional Java Programming](#)

Allen Holub

[Taming Java Threads](#)

Andrew S. Tanenbaum

Moderne Betriebssysteme, Hanser Verlag

Siehe auch: [Java threads: A comparative book review](#)

▲ Alte Klausuren

[Sommersemester 2002](#)

Service provided by [Mario Jeckle](#)

Generated: 2004-06-08T12:47:54+01:00

► [Feedback](#) ► [SiteMap](#)

► [This page's original location: http://www.jeckle.de/vorlesung/javaThreads/index.html](#)

► [RDF description for this page](#)

▲ **Home**

▶ **Dialog**

▲ **Fragen, Anregungen, Wünsche? Die Feedback-Seite!**

Hyperlinks

- ☐ Ungültiger Hyperlink ...
- ☐ Überflüssiger Hyperlink ...
- ☐ fehlender hyperlink ...

... von Seite:

... nach Seite:

Bemerkungen:

Seiteninhalt

- ☐ Nicht mehr aktueller Inhalt ...
- ☐ Inkorrektter Inhalt ...
- ☐ Irreführender Inhalt ...

... auf Seite:

Bemerkung:

Anregungen, Wünsche, Fragen

Bitte tragen Sie weitere Anregungen, Wünsche und Fragen hier ein:

Und nun...

Ihr Name:

Ihre Email-Adresse:

ab die Post!

Eingaben löschen

Service provided by [Mario Jeckle](#)

Generated: 2004-06-08T12:49:04+01:00

► [Feedback](#)

► [SiteMap](#)

► [This page's original location: http://www.jeckle.de/feedback.html](#)

► [RDF description for this page](#)

▲ Home

- ▼ **Unified Modeling Language (UML)**
- ▼ **eXtensible Markup Language (XML)**
- ▼ **XML Metadata Interchange (XMI)**
- ▼ **Web Services**
- ▼ **Vorträge und Publikationen**
- ▼ **Vorlesungen**
- ▼ **Studien- und Abschlußarbeiten**
- ▼ **Internet Search Engines / Suchmaschinen**
- ▼ **Mersennesche Primzahlen**
- ▼ **Web Services Workshop WS-RSD'02**
- ▼ **Free Stuff --- Software & Downloads**
- ▼ **RSS-Newsfeeds**

- ▶ **European Conference on Web Services 2004**
- ▶ **International Conference on Web Services Europe 2003**
- ▶ **Web Services @ Berliner XML-Tage 2004**
- ▶ **Web Services @ Berliner XML-Tage 2003**
- ▶ **XML-Strategie**
- ▶ **XML Acronym Demystifier Project**
- ▶ **Call for Papers Corner** **New**
- ▶ **GOOAL.net**
- ▶ **feedback**
- ▶ **Rotkreuz Mitgliederverwaltung**
- ▶ **Suchen ...**
- ▶ **Feedback**
- ▶ **Dialog ...**
- ▶ **Über diese Seiten ...**
- ▶ **Was gibt's hier Neues?**
- ▶ **RSS Newsfeed**
- ▶ **Mario Jeckle**
- ▶ **Mein PGP public key für Mailverschlüsselung, etc.**



Unified Modeling Language (UML)

- [UML Resource Center](#)
- [UML FAQs @ jeckle.de](#)
- [UML Links](#)
 - [Generelles](#)
 - [Tutorials](#)
 - [Frequently Asked Questions \(FAQ\) und Mailinglisten](#)
 - [UML @ OMG](#)
 - [Austauschformate](#)
 - [Werkzeugbezogenes ...](#)
 - [Objektorientierung allgemein](#)
 - [UML Autoren](#)
 - [UML auf gut Deutsch](#)(Vorschlag einer Gruppe deutschsprachiger Autoren zur einheitlichen deutschen Übersetzung der UML-Begriffe)
- [UML Bücher](#)
- [UML Official Specification Documents / Offizielle UML Spezifikation](#)
 - [General Information](#)
 - [UML Profiles und sonstige UML-bezogene Aktivitäten](#)
 - [UML v2.0](#)
 - [UML v1.5](#)
 - [UML v1.4](#)
 - [UML v1.3](#)
 - [UML v1.2](#)
 - [UML v1.1](#)
 - [Unterschiede zwischen den UML-Versionen](#)
- [UML Online Publications / Netz Ressourcen zur UML](#)
- [UML Tools](#)
- [UML2SVG-Transformationsdienst](#)



eXtensible Markup Language (XML)

- [XML Resource Center](#)
 - [Was ist die XML?](#)
 - [Links](#)
 - [Parser](#)
 - [XML-Editoren](#)
 - [XML \(Einsteiger-\)Informationen & Tutorials](#)
 - [Mailinglisten und User Groups](#)
 - [Konferenzen](#)
 - [FAQs und Antworten darauf](#)
 - [Vorlesung XML](#)
- [Entwurf von XML-Sprachen](#)
- [XML und Sicherheit \(Digitale Signatur, Verschlüsselung, ...\)](#)
- [Eine Übersicht existierender XML-Sprachen](#)
 - [Channel Description Format \(CDF\)](#)
 - [Chemical Markup Language \(CML\)](#)
 - [Customer Profile Exchange \(CPEX\)](#)
 - [Mathematical Markup Language \(MathML\)](#)
 - [Querying XML \(XQuery\)](#)
 - [RDF Site Summary \(RSS\)](#)
 - [Resource Description Framework \(RDF\)](#)
 - [Simple Markup Language \(SML\) und verwandte Ansätze \(TMML, Minimal XML, YAML, ...\)](#)
 - [Voice ML](#)
 - [Wireless Markup Language \(WML\)](#)
 - [XML Hypertext Markup Language \(XHTML\)](#)
 - [XML Linking Language \(XLink\)](#)
 - [XML Path Language \(XPath\)](#)
 - [XML Pointer Language \(XPointer\)](#)
 - [XML Protocols \(XML-RPC, SOAP, W3C's XML-Protocols, ...\)](#)
 - [XML Schema \(XSD, DCD, DT4DTD, Schematron, RELAX, ...\)](#)
 - [Projekt zur Übersetzung der XML-Schema-Spezifikation ins Deutsche](#)
 - [XSD-DE Chat vom 2001-08-16](#)
 - [XSD-DE Chat vom 2001-09-11](#)
 - [XSD-DE Chat vom 2001-10-08](#)
 - [XML Stylesheet Language \(XSL\)](#)
 - [XSL Transformations \(XSLT\)](#)
 - [Konversion nativer Formate nach XML](#)
- [Vorlesung eXtensible Markup Language](#)



XML Metadata Interchange (XMI)

- [XMI Resource Center](#)
- [Beispiele](#)
 - [Einfaches UML-Diagramm in XMI v1.0, v1.1 und als Rational Rose Export](#)
 - [Umfangreicheres UML-Diagramm in XMI v1.0](#)
 - [Generierung einer eigenen XMI-Sprache](#)
 - [Nutzung der MOF-DTD: UML v1.4 Metamodell in XMI v1.1](#)
 - [Generierung eines XSD-Schemas aus einem UML-Klassendiagramm unter Nutzung von XMI\[UML\]](#)
- [XMI News](#)
- [Werkzeuge mit XMI-Support](#)
- [Wozu ein Metadaten-Austauschformat?](#)
- [Ziele von XMI](#)
- [Hintergründe zur Entwicklung von XMI](#)
- [Offizielle Spezifikationsdokumente](#)
- [XMI Production for XML Schema](#)
- [Präsentationen, Vorträge und Veröffentlichungen](#)
- [Links](#)
- [Entwurf von XML-Sprachen mit XMI](#)
- [Modellaustausch mit dem XML Metadata Interchange Format](#)
- [UML2SVG-Transformationsdienst](#)



Vorträge und Publikationen

[Vorträge und Publikationen](#)

[Abstracts und Kurzfassungen](#)



Vorlesungen

- [Einführung in das .NET-Framework mit C#](#) 
- [Systemarchitekturen](#)
 - [Vorlesungsinhalt](#)

- [Mailingliste zur Vorlesung](#)
 - [Literatur](#)
 - [Aktuelles](#) **XML**
- [Standardisierung](#)
 - [Abstract](#)
 - [Ziel der Veranstaltung](#)
 - [Durchgeführtes Projekt](#)
 - [Links](#)
 - [Mailingliste zur Vorlesung](#)
 - [Aktuelles](#) **XML**
- [DB-Anwendungen](#)
 - [Vorlesungsinhalt](#)
 - [Mailingliste zur Vorlesung](#)
 - [Literatur](#)
 - [Links](#)
 - [Aktuelles](#) **XML**
 - [Alte Klausuren](#)
- [e-Business Engineering](#)
 - [Vorlesungsinhalt](#)
 - [Mailingliste zur Vorlesung](#)
 - [FAQs zur Vorlesung](#)
 - [Aktuelles](#) **XML**
 - [Alte Klausuren](#)
- [Integration Engineering](#)
 - [Vorlesungsinhalt](#)
 - [Mailingliste zur Vorlesung](#)
 - [Aktuelles](#) **XML**
- [Datenbanken](#)
 - [Vorlesungsinhalt](#)
 - [Mailingliste zur Vorlesung](#)
 - [Übungsaufgaben](#)
 - [FAQs zur Vorlesung](#)
 - [Aktuelles](#) **XML**
 - [Alte Klausuren](#)
- [eXtensible Markup Language](#)
 - [Teleteaching](#)
 - [Vorlesungsinhalt](#)
 - [Gliederung](#)
 - [Termine](#)
 - [FAQs zur Vorlesung](#)
 - [Anmeldung zum Praktikum im WS2002/03](#)
 - [Alte Klausuren](#)
 - [Übungsaufgaben:](#)
 - [1. Übungsaufgabe](#)
 - [2. Übungsaufgabe](#)
 - [Hinweise, Anmerkungen und FAQs zu dieser Übung](#)
 - [3. Übungsaufgabe](#)
 - [Hinweise, Anmerkungen und FAQs zu dieser Übung](#)
 - [4. Übungsaufgabe](#)
 - [Hinweise, Anmerkungen und FAQs zu dieser Übung](#)
 - [5. Übungsaufgabe](#)
 - [Hinweise, Anmerkungen und FAQs zu dieser Übung](#)
- [Alte Klausuren](#)
- [Script](#)
- [Hinweise zum Script](#)
- [Empfohlene Literatur](#)
- [Archiv: Praktikumsaufgaben und -Lösungen aus dem WS2001/02](#)

• [Java Threads](#)

- [Inhaltsübersicht](#)
- [Script](#)
- [Klausur vom Sommersemester 2002](#)
- [Literatur](#)
- [Internetressourcen zum Thema](#)

• [Java](#)

- [Script](#)
- [Gliederung](#)
- [Abstract](#)

• [Systemnahe Software](#)

• [Software Engineering I](#)

- [Hands on Java](#)
- [Objektorientierung](#)
- [Applikationserstellung](#)
- [Anwendungsaspekte](#)

• [Chat zur Vorlesung](#)

• [Vorlesungsevaluierung](#)



Studien- und Abschlubarbeiten

- [Mögliche noch zu vergebende Themen](#)
- [Gegenwärtig laufende Arbeiten](#)
- [Abgeschlossene Arbeiten](#)
- [HOWTO: Hilfestellung bei der Anfertigung von Arbeiten](#)



Web Services

- [Web Services -- eine technisch orientierte Einführung](#)
- [Sicherheitsaspekte XML-basierter Web-Services](#)
- [Web Service Workshop WS-RSD'02](#)
- [Web Services @ Berliner XML-Tage](#)



GOOAL.net

Deutsche Mailingliste rund um die Objekttechnologie

[direkt zu GOOAL.net ...](#)



Internet Search Engines / Suchmaschinen

- [Deutsche Suchmaschinen](#)
- [Meta-Suchmaschinen](#)
- [Internationale Suchmaschinen](#)
- [E-Mail Finder](#)
- [Dictionaries und Übersetzungshilfen](#)
- [File Finder](#)
- [Books & Journals](#)
- [Page Registrierung](#)



Mersennesche Primzahlen

- [Mersennesche Primzahlen](#)
- [SETI @ Home](#)
- [Rätsel der Proteinfaltung](#)



Web Services Workshop WS-RSD'02

- [Workshop Homepage](#)
- [Call for Papers](#)
- [Workshop Program, Schedule, and Presentation Slides Workshop Program, Schedule, and Presentation Slides](#)



Free Stuff --- Software & Downloads

- [XInclude Task for Ant](#)
- [Java version of Linpack benchmark](#)
- [Timer Task for Ant](#)
- [SOAPing](#) **New**
- [Java Assembler Playground](#) **New**



RSS-Newsfeeds

- [jeckle.de Newsfeed \(HTML-Version\)](#)
- [Linux Kernel News](#)
- [Call for Papers Corner \(HTML-Version\)](#)
- [DB-Anwendungen Newsfeed \(HTML-Version\)](#)
- [e-Business Engineering Newsfeed \(HTML-Version\)](#)
- [Datenbanken Newsfeed \(HTML-Version\)](#)
- [Mehr Informationen zu RSS](#)



Feedback

[direkt zu Seite ...](#)



Rotkreuz Mitgliederverwaltung

[direkt zu Seite ...](#)



Dialog

[direkt zu Seite ...](#)



Über diese Seiten ...

[direkt zu Seite ...](#)



Suchen ...

[direkt zu Seite ...](#)



Mario Jeckle ...

[direkt zu Seite ...](#)

[PGP public key](#)

Service provided by [Mario Jeckle](#)

Generated: 2004-06-08T12:51:56+01:00

► [Feedback](#)

► [SiteMap](#)

► [This page's original location: http://www.jeckle.de/sitemap.html](#)

► [RDF description for this page](#)