Java 2

new reference

Vorwort

Herzlich willkommen zu Java. Die Sprache und die gesamte Plattform Java haben sich seit ihrem glänzenden Start derweilen als feste Größe am Markt etabliert. Seit der Version 1.0 hat Java eine spannende Entwicklung durchgemacht und eine Vielzahl von Erweiterungen erhalten, welche das Leistungsvermögen immer weiter steigern. Vor allem kann man nun davon ausgehen, dass Kinderkrankheiten der ersten beiden Versionen 1.0/1.1 (vor allem der Version 1.0) der Vergangenheit angehören und Java damit zu einer äußerst leistungsfähigen Entwicklerplattform geworden ist.

Die kleine Flaute in der Java-Euphorie, welche nach dem fulminanten Start kurzzeitig eingesetzt hatte, ist definitiv überwunden. Dies belegen auch immer mehr zunehmende Anfragen an EDV-Unternehmen nach Kapazitäten für Java-Projekte. Das hat natürlich Konsequenzen. Überall finden sich Publikationen und Diskussionen über Java. Immer mehr erfahrene oder angehende Programmierer beschäftigen sich mit der neuen Technologie. Kaum ein ernst zu nehmendes Unternehmen im EDV-Markt kann darauf verzichten, Java als Basis für Projekte zu verwenden, Java-Programme oder -Dienstleistungen bereitzustellen oder zumindest die Entwicklungen rund um Java permanent zu verfolgen. Java-Kenntnisse sind also sehr gefragt und es steht zu erwarten, dass der Bedarf an Java-Programmierern weiter wachsen wird.

Vor allem ist Java aber spannend und leistungsfähig – die ideale Plattform, wenn man unbeschränkt von Restriktionen der auf der WINTEL-Plattform aufbauenden PCs selbst Anwendungen entwickeln möchte. Java kann die Basis für die plattformunabhängige Realisierung von grafischen Animationen, Spielen, interaktiven Online-Programmen und eigentlich allen denkbaren Programmen sein. Insbesondere Anwendungen in Netzwerken wie einem Internet/Intranet oder mit Datenbankzugriffen lassen sich mit Java ideal erstellen. Aber nicht nur Computer im engeren Sinn sind Zielplattform von Java – sämtliche Customergeräte mit integrierten Chips bieten sich als perfekte Zielplattform an. Vom Handy über den Videorekorder bis hin zu Geldkarten. Wenn die Visionäre Recht behalten, wird Java bereits in naher Zukunft in fast allen datenbasierenden Geräten zu finden sein. Java ist durch seine Anpassungsfähigkeit die exzellente Plattform für viele technische Geräte, welche irgendwelche Routinefunktionen bewerkstelligen müssen.

Ralph Steyer

WWW.RJS.DE

Einleitung

Zuerst soll geklärt werden, wie diese Referenz aufgebaut und was dort über Java zu finden ist. Das Buch besteht nachfolgend aus sechs weiteren zentralen Abschnitten:

- Was ist Java? Grundlagen. Im Abschnitt »Was ist Java? Grundlagen« sollen die wichtigsten Grundlagenbegriffe von Java in sehr kompakter Form durchgesprochen werden. Dies beinhaltet Hintergrundinformationen zu Java (sowohl technischer, als auch allgemeiner Natur), was in Java 2 neu ist und Informationen zum Java Development Kit.
- Die Java-Syntax. Dort werden die zentralen Begriffe von der übergeordneten Grundstruktur von Java besprochen, welche zusammen die Syntax der Sprache bilden. Das sind unter anderem Datentypen, Klassen, Objekte, Schnittstellen, Ausnahmen, Variablen, Methoden, Zuweisungen, Ablaufsteuerung, und einige mehr. Dabei soll dieses Kapitel einen grundsätzlichen Überblick über das Javakonzept bieten, indem die einzelnen Begriffe strukturiert analysiert werden.
- Die Java-Schlüsselworte. Dieser Abschnitt ist von zentraler Bedeutung, denn eine Programmiersprache besteht in der Grundstruktur aus einer gewissen Anzahl von Buchstabenfolgen, die neben der Sprachgrammatik der wesentliche Teil der Sprachdefinition sind. Diese sogenannten Schlüsselworte haben eine definierte Bedeutung und bilden den Kern einer Programmiersprache, die einem Programmierer auf jeden Fall bekannt sein muss. Jedes Schlüsselwort von Java 2 wird in diesem Abschnitt beschrieben.
- Die Pakete der Java-2-Plattform. In dem Bestandteil der Referenz wird das aktuelle Standard-API von Java besprochen. In einem allgemeinen Überblick wird so die bereits standardmäßig vorhandene Leistungsfähigkeit des Java-API dargestellt. Dieser Abschnitt ist auch deshalb interessant, weil sich nur so die ganzen Erweiterungen / Veränderungen / Verlagerungen von Java über die verschiedenen Versionen nachvollziehen lassen. Daneben soll er als Ergänzung zu der frei verfügbaren Online-Dokumentation des Java-API verstanden werden und bietet dazu erweiternden Nutzwert.
- Die als deprecated erklärten Elemente des Java-API 1.2. Das Java-API hat sich in der Version 1.2 (dem API der Java-2-Plattform) erheblich gegenüber seinen Vorgängern verändert. Um die Veränderungen in dem Java-API verfolgen zu können, benötigt man eine Auflistung der als deprecated gekennzeichneten Elemente (deprecated können Sie sich erst einmal als »veraltet« vorstellen – auf diese Thematik gehen wir aber an besagter Stelle intensiv ein). Dabei werden in der Auflistung in diesem Abschnitt alle Klassen, Schnittstellen, Ausnah-

men, Variablen, Methoden und Konstruktoren angegeben, sowie optional einige Informationen darüber, wann die Elemente als deprecated gekennzeichnet wurden und welcher Ersatz dafür verwendet werden kann.

- Der Index. Ein äußerst umfangreicher Index ermöglicht das schnelle Auffinden der zentralen Java-Stichworte sowie von den aktuellen Paketen, Klassen, Schnittstellen, Ausnahmen und Errors.

Was ist Java? Grundlagen

Java ist eine weitgehend betriebssystem- und hardwareunabhängige Sprache beziehungsweise ganze Entwicklungsplattform, deren Fokus zum einen auf dem Internet/Intranet, zum anderen in dem Bereich der datenbasierenden Konsumerelektronik liegt. Java ist also direkt für eine extrem heterogene Welt entwickelt worden. Zusätzlich ist Java objektorientiert, was das zweite wichtige Kriterium ist. Daneben gilt Java als sehr netzwerksicher und dadurch geeignet für Anwendungen, die in einer heterogenen Netzwerkumgebung ablaufen sollen. Gerade dort wird hohe Stabilität gefordert und auch diese stellt Java über zahlreiche Mechanismen sicher.

Entwickelt wurde Java von Sun Microsystems. Mittlerweile hat Sun die offizielle Verantwortung zwar auf die Firma JavaSoft – eine direkte Tochterfirma – übertragen, aber wir werden im Folgenden nicht so genau zwischen Sun (`http://java.sun.com`) und JavaSoft (`http://java.javasoft.com`) trennen.

Der Java-Ansatz

Java-Quellcode ist auf den ersten Blick dem Quellcode der Sprachen C und C++ sehr ähnlich, weist gegenüber C/C++ jedoch nach Ansicht vieler Fachleute zahlreiche Verbesserungen auf. Java-Quellcode wird durch den Java-Compiler in ein plattformunabhängiges Bytecode-Format übersetzt. Dieser wird auf der Zielmaschine durch die sogenannte Java Virtual Machine (JVM) interpretiert, einem virtuellen Computer, welcher nur im Speicher eines Rechners oder Customergerätes zur Laufzeit resistent vorhanden ist. Die JVM ist das Herz von Java und muss auf jedem Rechner oder auch Customergerät, wie einem Java-Videorekorder, vorhanden sein, welcher(s) Java-Anwendungen ausführen möchte (dies ist die einzige Einschränkung der Plattformunabhängigkeit, weshalb wir im Folgenden nur noch von Plattformunabhängigkeit sprechen wollen). Die virtuelle Maschine stellt eine Abstraktionsschicht zwischen dem kompilierten Bytecode und der zugrunde liegenden Hardwareplattform und dem Betriebssystem dar und realisiert die Umsetzung des Bytecodes in ausführbare Prozessorbefehle. Die Java Virtual Machine kennt die Spezifika der jeweiligen Hardware, auf der sie läuft, und kann so auf die Eigenheiten eines bestimmten Prozessortyps und der restlichen Hardware Rücksicht nehmen. Java-Applikationen werden also nicht mehr zur Kompilierzeit für eine Prozessorfamilie generiert, sondern in ganz allgemeinen Code übersetzt. Dennoch können sie die jeweils spezifischen Eigenschaften eines bestimmten Prozessors nutzen, indem sie über die implementierte JVM-Schnittstelle darauf zugreifen.

Die virtuelle Maschine kümmert sich auch um grundlegende Java-Operationen wie die Objekterstellung und die Müllbeseitigung (sprich die Speicherfreigabe). Java besitzt eine integrierte Speicherverwaltung in Form des Garbage Collectors. Die Implementierung der Speicherverwaltung ist in der Regel unter Java nicht Aufgabe eines Programmierers. Wenn ein Objekt nicht mehr referenziert wird (also nicht mehr benötigt wird), wird der belegte Speicher automatisch freigegeben. Der Garbage Collector läuft bei jeder aktiven Java-Applikation immer als eigenständiger Thread mit niedriger Priorität im Hintergrund (auch wenn das Programm/Applet nicht explizit als Multithreading-Applikation gekennzeichnet ist): Das Freigeben von Ressourcen findet meist statt, wenn das System wenig belastet ist.

Die JVM kümmert sich gleichfalls darum, dass zur Laufzeit benötigte Informationen bereitstehen. Das Java-Konzept stellt sicher, dass Programmbibliotheken zur Laufzeit dynamisch von einer beliebigen verfügbaren Maschine (auch aus einem Netzwerk) geladen werden. Eine Java-Applikation kann dadurch recht klein gehalten werden.

Die offizielle Definition von Sun Microsystems für Java lautet folgendermaßen:

Java: eine einfache, objektorientierte, dezentrale, interpretierte, stabil laufende, sichere, architekturneutrale, portierbare und dynamische Sprache, die Hochgeschwindigkeitsanwendungen und Multithreading unterstützt.

Im Einzelnen bedeutet dies:

- Einfach: Obwohl Java sehr nahe an der komplizierten C/C++-Syntax konzipiert wurde, verzichtet Java auf viele C/C++-Schwächen und vereinfacht zahlreiche Vorgänge. Java hat unter der Prämisse »weniger ist mehr« erheblich gewonnen.
- Klein: Die Größe des Basisinterpreters und die Klassenunterstützung betragen in etwa 40 Kbyte RAM. Zusätzliche Standardbibliotheken und Thread-Unterstützung benötigen weitere 175 Kbyte.
- Objektorientiert: Java ist streng objektorientiert. Daten werden immer als Objekte definiert und Methoden verwendet, um diese Objekte zu bearbeiten. Java verwendet Vererbung, jedoch keine Mehrfachvererbung.
- Dezentral: Java erfüllt ein wesentliches Charakteristikum von Client-Server-Anwendungen – die Fähigkeit, Informationen und die Last für die Verarbeitung der Daten zu verteilen.
- Interpretiert: Wie oben bereits erwähnt, wird der Java-Quellcode in einen plattformunabhängigen Bytecode kompiliert. Dieser Bytecode wird dann auf der Zielmaschine von der JVM (etwa als Bestandteil eines Browsers, wenn es sich um ein Applet handelt) interpretiert, d.h. während der Ausführung in den jeweiligen Maschinencode übersetzt. Java gilt aufgrund der Aufteilung als interpretiert und kompiliert zur gleichen Zeit. Diese Vorgehensweise ermöglicht viele besondere Eigenschaften von Java, beispielsweise die Portierbarkeit oder den hohen Sicherheitsstandard.

- Stabil: Je stabiler eine Sprache ist, desto unwahrscheinlicher ist es, dass in dieser Sprache geschriebene Programme abstürzen. Ein wesentliches Kriterium für die Stabilität einer Programmiersprache ist, dass bereits während der Kompilierungsphase der größte Teil der Datentypüberprüfung ausgeführt wird (der Fachbegriff dafür ist stark typisiert), nicht erst zur Laufzeit wie bei nicht so stabilen Sprachen. Eine stark typisierte Sprache wie Java ermöglicht also bereits ein intensives Überprüfen des Codes während der Kompilierungsphase. Ein weiteres Stabilitätskriterium von Java-Programmen ist, dass sie keinen Zugriff auf den vollständigen Speicherbereich eines Computers – insbesondere den Systemspeicher – haben. Dies gilt besonders für Applets. Java verfügt über eine eingebaute Begrenzung der Zugriffsmöglichkeiten. Damit verringert sich die Absturzwahrscheinlichkeit eines Java-Programms erheblich. Ein weiterer Stabilitätsaspekt von Java ist die Sicherheitsüberprüfung durch einen Linker, welcher Teil der Laufzeitumgebung ist.
- Sicher: Die Unterbindung von beliebigen Speicherzugriffen durch Java ist nicht nur ein Stabilitätskriterium, sondern trägt gleichfalls zur Sicherheit bei. Da Java-Programme zuerst in die besagten Bytecode-Anweisungen übersetzt werden, können diese Anweisungen relativ komfortabel überprüft werden (Bytecode ist sogar mit einem Editor – insbesondere einem Hex-Editor – noch teilweise lesbar). Bytecode-Anweisungen sind leicht zu kontrollieren, nicht plattformspezifisch und enthalten zusätzliche Typinformationen. Diese können zur Überprüfung des legalen Status und von möglichen Sicherheitsverletzungen verwendet werden.
- Architekturneutral: Java ist auf verschiedenen Systemen mit unterschiedlichen Prozessoren und Betriebssystemarchitekturen lauffähig. Der kompilierte Java-Bytecode kann auf jedem Prozessor ausgeführt werden, welcher einen Java-fähigen Browser beziehungsweise die virtuelle Maschine von Java im Allgemeinen unterstützt. Plattformabhängiger Binärcode wird nicht direkt erzeugt, sondern der Java-Bytecode wird während der Laufzeit in systemeigenen Maschinencode übersetzt.
- Portierbar: Java ist portierbar. Implementationsabhängige Aspekte fallen bei Java im Wesentlichen durch die exakt in der Länge festgelegten Datentypen weg. In Java wird die Größe einfacher Datentypen genau spezifiziert, und auch, wie sich die Arithmetik gegenüber diesen Datentypen verhält (unabhängig von der Plattform!!). Java-Datentypen unterscheiden sich dabei in einigen Details von gewohnten Datentypen aus vielen anderen Programmiersprachen. Im wesentlichen in der Länge (besonders auffällig ist der `char`-Typ, welcher nicht wie üblich aus einem Byte besteht), aber auch in ein paar weiteren Einzelheiten. Es gibt keinen primitiven Datentyp für Felder (Arrays). Dies liegt daran, dass Arrays in Java wie Objekte behandelt werden (eine zwingende Folge der strengen Objektorientierung von Java) und keine eigene Darstellung

besitzen. Dafür besitzen sämtliche Datentypen außer `char` und `boolean` ein Vorzeichen. Die JVM (Sun) besitzt die folgenden Typen primitiver Daten:

  - `boolean`, 1 Bit
  - `byte`; 8 Bit
  - `short`; 16 Bit
  - `int`; 32 Bit
  - `long`; 64 Bit
  - `float`; 32 Bit
  - `double`; 64 Bit
  - `char`; 16 Bit
- Leistungsfähig: Die Geschwindigkeit von interpretiertem Java-Bytecode ist für die meisten Aufgaben ausreichend. Die Java-Version 2 beinhaltet diverse Neuerungen zur Steigerung der Performance. Außerdem kann nativer Code integriert werden (wovon jedoch aus Sicherheitsgründen dringend abzuraten ist).
- Unterstützung von echtem Multithreading: Java erlaubt es, dass mehrere Aufgaben oder Prozesse quasi gleichzeitig ausgeführt werden. Die Vorteile (bessere interaktive Antwortfähigkeit, besseres Laufzeitverhalten) hängen jedoch stark von der zugrunde liegenden Plattform ab. Java kann insbesondere auf Multiprozessorumgebungen seine Stärken ausspielen.
- Dynamisch: Java kann sich über das Paketkonzept an eine sich ständig weiterentwickelnde Umgebung anpassen und wird nicht von fremden Klassenbibliotheken abhängig.

Applets vs Stand-alone-Applikationen

Ein oft gemachter Fehler bei Java-Laien besteht darin, Java-Applikationen und Java-Applets gleichzusetzen, aber man muss zwischen Java-Applets und Java-Applikationen unterscheiden.

Java-Applets sind kleine Java-Anwendungen, welche nur innerhalb eines Webbrowsers oder des Appletviewers ausgeführt werden können. Über eine entsprechende Referenz innerhalb einer HTML-Seite mit einem speziellen Tag – in der Regel dem `APPLET`-Tag oder seit HTML 4.0 dem `OBJECT`-Tag – erfolgt die Einbettung in die Webseite. Ein Applet kann dabei über ein Netzwerk auf die lokale Maschine geladen und dort ausgeführt werden, aber auch vom lokalen System selbst kommen. Applets, die über einen nichtlokalen Pfad (etwa ein offenes Netzwerk, aber auch ein als unsicher eingeschätzter Pfad von der lokalen Maschine) geladen werden, haben in der Regel aus Sicherheitsgründen eingeschränkte Zugriffsmöglichkeiten auf Systemressourcen. Die Art der Einschränkungen ist im Browser beziehungsweise Appletviewer konfigurierbar.

Stand-alone-Applikationen sind eigenständig lauffähige Programme, die im Gegensatz zu den Applets keinen Container wie einen Browser mehr benötigen. Sie laufen direkt in der JVM und haben dementsprechend weitergehende Möglichkeiten auf dem lokalen System. Insbesondere benötigen eigenständige Java-Applikationen immer eine `main()`-Methode (die zentrale Methode, über die Einstieg und Ablauf eines Programms gesteuert werden). Eine solche `main()`-Methode gibt es bei einem Applet nicht.

Hintergründe zur JVM und dem Compiler

Die Grammatik von Java ist so aufgebaut, dass Programme robust und sicher sind und auch schon viele Fehler bei der Programmierung abgefangen werden. Insbesondere besitzt das Java-System einige Eigenschaften, die sicherstellen, dass Programmcode nicht die Sicherheit des Systems kompromittiert. Dies gilt sowohl zur Kompilierzeit als auch zur Laufzeit.

Kompilierzeit

Zur Kompilierzeit wird beispielsweise sichergestellt, dass Objekte nicht als Objekte eines anderen Typs verwendet werden und alle eventuellen Typumwandlungen legal sind. Ebenso wird sichergestellt, dass `final`-deklarierte Variablen (die Java-Konstanten) oder andere finale Elemente nicht verändert werden. Und nicht nur das – jede Variable in Java hat immer einen definierten Typ und Wert. Bedingt durch die Eigenschaften der Sprache ist die Belegung des Maschinenstapels (Stacks) an jeder Stelle des Programms zur Kompilierzeit bekannt. Stacküberläufe werden zur Kompilierzeit erkannt. Daneben erfolgt eine Typüberprüfung bereits auf dem Stack.Jede Methode wird bereits beim Kompilieren auf eventuell mögliche checked Exceptions (Ausnahmen) geprüft, sofern diese innerhalb einer `throws`-Anweisung in der Deklaration einer Methode genannt werden. Die Fähigkeit von Methoden, bestimmte Exceptions zu erzeugen, muss entweder dokumentiert oder weitergereicht werden. Dies stellt sicher, dass potentielle Ausnahmen auf jeden Fall behandelt werden. Das kann eventuell auch nur durch ein globales Management erfolgen, das nicht spezifisch auf die Eigenheiten der Exception eingeht. Neben den checked Exceptions gibt es die unchecked Exceptions. Diese müssen nicht individuell behandelt werden, sondern deren Behandlung obliegt dem System. Zu den unchecked Exceptions gehören die Elemente der Klasse `RuntimeException`, ihrer Unterklassen und der Klasse `Error` und deren Unterklassen. Ausdrücke, die Variablen oder Felder initialisieren, dürfen keine Exceptions erzeugen. Das wird ebenfalls zur Kompilierzeit sichergestellt.

Laufzeit

Zur Laufzeit einer Java-Applikation/Applets erfolgen zahlreiche weitere Checks durch die Java Virtual Machine. Diese Überprüfungen fangen viele der potentiellen Probleme ab, wie sie etwa bei der verwandten Sprache C/C++ auftreten können. Um ein Beispiel zu nennen – es werden permanent Arrays überprüft, sofern darauf ein Zugriff erfolgt. Arrays sind (wie wir schon etwas weiter oben festgehalten haben) in Java Objekte – eine zwingende Folge der strengen Objektorientiertheit von Java und ein massiver Unterschied zu C/C++. Jeder Zugriff auf ein Arrayelement wird zur Laufzeit geprüft: Ist der Index negativ oder jenseits der Grenze des Arrays, wird eine Exception ausgelöst. Arrays haben eine feste Länge, die während der Programmausführung nicht geändert werden kann.

Ein anderes C/C++ – Problem sind Zugriffe auf Strings, wenn man dort die Länge des `Strings` nicht korrekt beachtet. Java-Strings sind Objekte der Klasse `String` und haben eine wohldefinierte, feste Länge. Referenzen auf Zeichen hinter dem letzten Zeichen eines Strings werden zur Laufzeit entdeckt und abgefangen.

Bereits beim Laden eines Java-Programms werden von der JVM zahlreiche Überprüfungen aufgerufen – durch den Bytecode-Verifier und den Classloader. Dies bedeutet, dass jeglicher Java-Bytecode beim Laden in den Speicher durch spezielle Mechanismen analysiert wird. Dabei unterscheidet Java wie oben angedeutet den Herkunftsort des Bytecode. Java-Bytecode, der über einen als unsicher eingestuften Zugriffspfad geladen wird, gilt selbst prinzipiell als unsicher und wird in verschiedenen Stufen analysiert. Damit wird verhindert, dass Programme ausgeführt werden, welche die Sicherheit des Systems gefährden. Die Prüfungen werden beim Laden eines Programms durch den Classloader und vor der Ausführung eines Programms durch den Bytecode-Verifier durchgeführt. Insbesondere unterliegen Applets per Voreinstellung starken Einschränkungen – etwa bei Zugriffen auf ein Netzwerk durch ein Applet, die je nach der Konfiguration des Containers[1] (etwa ein Browser) eingeschränkt werden können. Applets, die über ein Netzwerk beziehungsweise einen lokalen, aber als unsicher deklarierten Bereich geladen werden, ist normalerweise der Zugriff auf bestimmte Systemressourcen verboten. Sie dürfen beispielsweise keine Dateien auf der lokalen Maschine lesen oder schreiben, eigenständig Netzwerkverbindungen eröffnen, außer zu der Maschine, von der der Appletcode geladen wurde, Programme starten oder native Methoden aufrufen.

Wie bereits der Compiler stellt auch der Bytecode-Verifier zur Laufzeit sicher, dass Objekte nur als Objekte ihres spezifischen Typs angesprochen werden und dass keine illegalen Typumwandlungen stattfinden. Hierbei wird auch der Stack geprüft. Ebenso wird überprüft, dass die durch die Zugriffsmodifier (Zugriffsmodifizierer) `public`, `private` und `protected` spezifizierten Einschränkungen in den Zugriffsmöglichkeiten auf Objekte, ihre Felder und Methoden eingehalten werden.

Ein weiterer Aspekt ist, dass Java sowohl die OO-Technik (OO steht für Objektorientierung oder objektorientiert) des Überschreibens, als auch das Überladens unterstützt. Damit lassen sich Methoden, Variablen und Klassen verdecken. Java-Klassen, die von verschiedenen Maschinen geladen werden, können einander jedoch nicht verdecken, da für jede Netzwerkmaschine und für die lokale Maschine eigene Namensräume existieren.

1. **Container im Sinne einer Laufzeitumgebung für Applets.**

Was ist neu in Java 2.0?

Erheblich verspätet erschien zum Jahreswechsel 1998/99 Java 2.0. Zuerst wurde die lange angekündigte und mehrfach verschobene Finalversion des JDK 1.2 (Java Developer Kit) – der Java-Entwicklungsumgebung – freigegeben. Kurz danach gab Sun auf der Java Business Expo bekannt, dass es nicht nur ein neues JDK, sondern ein vollständiges Plattform-Update unter dem Namen Java 2.0 gibt.

Der Name überraschte doch ziemlich, denn es schien bis dahin klar, dass das vollständige Update der bis dahin vertriebenen JDK-Finalversion 1.1 und seiner ergänzenden Tools unter der Versionsbezeichnung Java 1.2 in Umlauf kommt. Darauf deuteten hauptsächlich die lange vorhandenen Betaversionen Java (1.2.1 bis 1.2.4) hin. Verwirrend ist gleichermaßen, dass das Java Development Kit – Kern der Java-2.0-Plattform – innerhalb der Java-2.0-Plattform als JDK 1.2 geführt wird. Die Veränderungen des Java-API, welche sich noch zwischen den Betaversionen 1.2 und der als Final freigegebenen Version ergeben haben, erzwingen einen solche Namenspolitik jedoch (beispielsweise beinhalten die ersten drei Betaversionen des 1.2-API und das API der Java-2-Plattform in vielen Bereichen große Abweichungen).

Wesentlicher Unterschied zwischen der vollständigen Java-Plattform 2.0 und dem JDK 1.2 ist, dass die Plattform zahlreiche weitere Programme, Tools und Konzepte enthält. Etwa das Java Servlet Development Kit zur Erstellung von Java-Servlets. Dieses wurde bis zur Beta-4-Version des JDK noch zu dem JDK gezählt, gilt jedoch in der Java-2-Plattform als eigenständiges Produkt. Oder auch andere Toolpakete wie das Java Media Framework, welche die Funktionalität des JDK in Bezug auf Multimedia erweitern. Die Java-2-Plattform mit dem JDK 1.2 beinhaltet unter anderem folgende Konzepte:

- Sichere und signierte Applets
- Collections Framework
- JavaBeans
- Internationalization
- Networking
- Reflection
- Package Version Identification
- Sound und Multimedia
- Reference Objects
- Object Serialization
- Innere Klassen
- Java Archive (JAR) Files
- Java Native Interface (JNI)
- Java Foundation Classes (JFC)

- Abstract Window Toolkit (AWT)
- Swing Components
- 2D Graphics and Imaging
- Input Method Framework
- Accessibility
- Drag&Drop-Datentransfer
- Interface Definition Language (IDL)
- Remote Method Invocation (RMI)
- Java Database Connectivity (JDBC)

Die wesentlichen Neuerungen der Java-2-Plattform gegenüber den Vorgängerversionen konzentrieren sich auf die Bereiche Sicherheit, Interoperabilität mit anderen Systemen, Plattformneutralität, Geschwindigkeitssteigerung, Internationalisierung und Vereinfachung der Entwicklungstätigkeit. Leider fanden viele Veränderungen im Laufe der vier Betaversionen des JDK 1.2 statt, weshalb diese Betaversionen in vielen Bereichen nicht mehr mit dem Finalrelease übereinstimmen. Gerade der Wechsel von der Beta-3-Version 1.2 auf die wenig beachtete Beta-4-Version 1.2 beinhaltete zahlreiche und sehr massive Modifikationen, weswegen viele Umsteiger von einem JDK 1.1.x oder einer der drei ersten Betaversionen 1.2 ziemliche Umstellungsprobleme haben dürften.

Kompatibilität & Inkompatibilität

Die Java-2-Plattform besitzt in der Sicherheitsfunktionalität eine gewisse Inkompatibilität zu den Vorgängern. Das Java-Sicherheitsprogramm der JDK-Version 1.1 `javakey` wurde durch die Programme `keytool` und `jarsigner` abgelöst. Daneben kamen die Sicherheitsprogramme `jar` und `policytool` hinzu. Das JDK 1.2 erweitert zudem den Umfang der Tools, welche von dem Sicherheitsmodell von Java betroffen sind, in der Finalversion um ein weiteres Programm (`oldjava`). Dies ist zwar nur ein Interpreter, aber dessen Hauptfunktion ist es, Applikationen zu unterstützen, welche über einen eigenen Security Manager verfügen, der auf dem originalen Sicherheitsmodell von Java 1.0/1.1 basiert. Dieser wird unter der Java-2-Plattform eine Exception auswerfen und nicht starten. Solche Applikationen können mit dem Programm `oldjava` als Interpreter jedoch weiter verwendet werden.

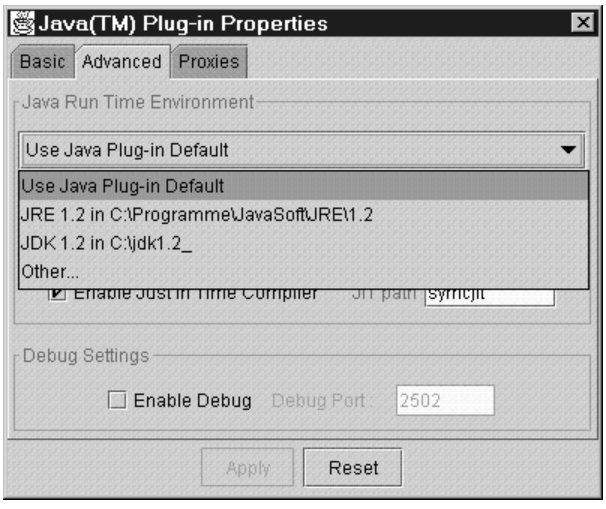
Im Allgemeinen gilt aber, dass Entwicklungen, welche mit einer älteren Version von Java erstellt wurden, ebenso in der virtuellen Maschine der Folgeversionen ohne Probleme laufen sollten. Dies gilt sowohl auf binärer Ebene, als auch auf Ebene des Quelltextes (allerdings definitiv ohne Gewährleistung durch Sun). Ausnahmen sind explizit diejenigen Java-Programme, welche in einer der Java-Vorgängerversionen erstellt wurden und die auf Klassen zurückgreifen, in welchen sicherheitsrelevante Löcher festgestellt wurden. Auch einige Java-Programme, die auf Klassen mit Implementations- oder Designbugs zurückgreifen, können unter den

neuen Versionen nicht mehr laufen. Solche Applikationen müssen bei einer Portierung von älteren JVM auf eine neuere JVM neu kompiliert werden.

Bei der Neukompilierung von älterem Java-Quellcode werden dann ziemliche Schwierigkeiten auftreten, wenn dort Java-Pakete importiert werden, welche es in dem neuen Konzept nicht mehr gibt (das ist zwar selten, aber möglich) oder die verlagert und/oder umbenannt wurden (das ist leider sehr oft der Fall). Beispielsweise sind die Pakete des gesamten Swing-Konzeptes gegenüber der bisherigen Konzeption (Version 1.1 und sogar Beta 1.2) vollständig verlagert und umbenannt worden. In solchen Fällen muss bei einer Neukompilierung in dem Java-Quelltext die `import`-Struktur dem neuen API natürlich angepaßt werden.

Eine gewisse Schwierigkeit bei Java-Applets ist die Integration von Java-Versionen jenseits von Java 1.0.2 in die gängigen Browser, welche oft nur Java 1.0.2 oder maximal 1.1.3 verstehen. Dies bedeutet nicht unbedingt, dass Applets mit einem alten JDK erstellt werden müssen. Soweit Applets mit einem beliebigen JDK erstellt und kompiliert wurden und nur auf Java-Standard-1.0.2 basierende Funktionalitäten nutzen, sollten sie auch in nur diesen Standard unterstützenden Browsern/Containern dargestellt werden können. Diese Abwärtskompatibilität wird allerdings von Sun nicht ausdrücklich garantiert.

Die Verwendung von alten JVMs in Browsern ist außerdem nicht mehr zwingend. Die Java-2.0-Plattform beugt – im Gegensatz zu den Vorgängerversionen – vor und beinhaltet ein Java-Plug-In für Webbrowser (ehemals bekannt als »Activator«). Damit ist im Prinzip immer eine vollständig zu einer beliebigen JVM kompatiblen Laufzeitumgebung für Applets vorhanden. In dem als Laufzeitmanager konzipierten Tool kann man in dem Registerblatt `ADVANCED` verschiedene Java-Laufzeitumgebungen einstellen, welche als Java-Plug-In für Browser fungieren.



Das neue Java-Plug-In des JDK 1.2.

Es ist also nicht nur möglich, zwischen verschiedenen auf dem Rechner installierten Java-Laufzeitumgebungen zu wechseln. Es kann jederzeit die neueste Java-Laufzeitumgebung auf einem Rechner installiert werden und diese steht dann denjenigen Browsern zur Verfügung, welche eine externe Java-Laufzeitumgebung verwenden können.

Hinweis: Leider funktioniert das Java-Plug-In nicht bei allen Plattformen/Browsern und man kann sich nicht unbedingt darauf verlassen, dass die Browser mit dem 1.2-API oder sogar bereits früher realisierten Techniken wie Java2D oder Swing tatsächlich zurechtkommen. Insbesondere Sicherheitsrestriktionen der Browser hebeln neuere Techniken immer wieder aus – trotz aktivierter 1.2-Java-Laufzeitumgebung. Ausführlichere Details und laufend aktuelle Informationen gibt es auf der Webseite »`http://java.sun.com/products/jdk`«**.**

Eine der gravierendsten Veränderungen im Hintergrund von Java betrifft die Verwendung der Laufzeitumgebung von Java. War es in den bisherigen Versionen (bis zu der Version 1.1 des JDK bzw. den meisten Betaversionen des JDK 1.2) im Wesentlichen ein Zugriff auf eine Datei namens `classes.zip`, hat sich in der Finalversion 1.2 der Zugriff auf die Laufzeitumgebung von Java erheblich verändert. Bei der Datei `classes.zip` handelt es sich um das RUNTIME-Modul (die Systemklassen) von Java in den damaligen Versionen. Die Datei enthält den vollständigen, kompilierten Code von Java und muss in diesen Versionen des JDK einigen JDK-Tools (insbesondere dem Compiler und dem Interpreter) zur Laufzeit zur Verfügung stehen. Das JDK 1.2-Final benötigt die Datei `classes.zip` gar nicht mehr (und stellt sie auch nicht mehr zur Verfügung – Ersatz ist im Wesentlichen die Datei `rt.jar`). Diese war vorher zwingend notwendig, aber die Laufzeitumgebung des JDK 1.2 hat sich vollständig geändert. Das wird insbesondere für solche Applikationen ein Problem, die `classes.zip` hartkodiert verwenden.

Die Veränderung des Zugriffs auf die Laufzeitumgebung hat Konsequenzen. Das JDK 1.2 verwendet für den Zugriff auf die Systemklassen nun einen »bootstrap class loader«, welcher seinen eigenen Pfad zur Suche der Systemklassen verwendet. Die Folge ist, dass im neuen JDK auch nur noch selbst erstellte Klassen bei Bedarf in den mit der Angabe `CLASSPATH` deklarierten Suchpfad aufgenommen werden müssen. Bei der Angabe `CLASSPATH` handelt es sich um eine Pfadangabe, über die Java (genau genommen bestimmte Tools von Java) Systemklassen sucht, die es zur Laufzeit benötigt. Es gibt den Suchpfad in zwei Varianten.

Einmal gibt es eine Umgebungsvariable dieses Namens, daneben gibt es einen weiteren Suchpfad, welcher als Option bei den entsprechenden Tools individuell gesetzt werden kann und die globale Angabe überschreibt. In den bisherigen Versionen des JDK wurden alle Java-Klassen aufgrund des Pfades gesucht, welcher über die Option `-classpath` bei einem Tool oder der globalen `CLASSPATH`-Umgebungvariable (über den `SET`-Befehl) gesetzt war. Sun empfiehlt nun prinzipiell für das

neue JDK, die globale Angabe überhaupt nicht mehr zu setzen und die individuelle Variante zu verwenden, sofern Klassen nicht bereits ohne die Angabe gefunden werden[2].

Der Grund liegt darin, dass das JDK 1.2 eine neue Art und Weise verfolgt, wie Klassen gefunden werden. Leider dokumentiert das Sun nicht so deutlich und leicht verständlich, wie es sicher wünschenswert wäre. Tatsächlich benötigen die meisten Tools des JDK die Systemklassen immer noch, aber nicht mehr die Angabe `CLASSPATH`, um sie zu finden. Die Systemklassen (in der Datei `jr.jar`) finden sie nach einer erfolgreich durchgeführten Installation des JDK automatisch, wenn sie im `lib`-Verzeichnis von `jre` (der Laufzeitumgebung) vorhanden sind. Dabei ist es belanglos, ob es sich um das `jre`-Verzeichnis innerhalb des JDK-Verzeichnisses handelt oder das eigenständige `jre`-Verzeichnis, welches im Setup zusätzlich mitinstalliert wird.

2. **In der Tat sind Fälle bekannt, wo ein auf ein falsches Verzeichnis gesetzter Suchpfad zu Problemen geführt hat. Praktische Erfahrungen des Autors müssen es ebenso bestätigen.**

Das JDK

Das Java Development Kit (JDK) ist die Sun-Entwicklungsumgebung zum Erstellen von Java-Applikationen, -Applets und -Komponenten. Es ist Kern sämtlicher Java-Entwicklungswerkzeuge (gleichfalls von Fremdherstellern). Das JDK wird in der aktuellen Version 1.2 (das Entwicklungspaket der Java-2-Plattform) von Sun derzeit für die folgenden Plattformen frei zur Verfügung gestellt:

- Das Win32-Release für Windows 95, Windows 98 und Windows NT auf Intel-basierender Hardware. Windows NT wird nur ab der Version 4.0 unterstützt. Als Minimalhardware fordert Sun einen 486/DX-Prozessor und 48 Mbyte RAM.
- Das Solaris/SPARC-Release. Nur die Solaris-Versionen 2.5.1, 2.6 und 7 (auch als 2.7 bekannt) werden unterstützt. Als Hauptspeicher werden 48 Mbyte gefordert. Unter Umständen werden Solaris-Patches benötigt.
- Das Solaris/Intel-Release. Nur die Solaris-Versionen 2.5.1, 2.6 und 7 (auch als 2.7 bekannt) werden unterstützt. Als Minimalhardware fordert Sun einen 486/DX-Prozessor und 48 Mbyte RAM. Unter Umständen werden Solaris-Patches benötigt.

Ältere Versionen des JDK gibt es zusätzlich für Macintosh System 7.5 für 68030, 68040 und PowerPC-basierte Macs sowie Sun Solaris 2.3 und 2.4 für SPARC-basierte Maschinen. Genauere Informationen (insbesondere zu den Patches) finden Sie unter `http://java.sun.com/products/jdk/faq.html`, das JDK selbst gibt es unter `http://java.sun.com`.

Das JDK besteht aus einer ganzen Reihe von Tools. Diese sind weitgehend Befehlszeilen-orientiert (der Appletviewer ist eine kleine Ausnahme) und erreichen damit natürlich nicht den Komfortstandard integrierter Entwicklungsumgebungen (IDE). Die JDK-Werkzeuge (den Compiler eingeschlossen) können über eine einfache Befehlszeileneingabe auf Prompt-Ebene aufgerufen werden. Der Java-Source selbst (eine Datei mit Erweiterung `.java`) kann mit jedem einfachen ASCII-Text-Editor erstellt und bearbeitet werden.

Die wichtigsten Tools waren schon in den 1.0.x-Versionen dabei, viele kamen in den 1.1.x-Versionen hinzu und auch die Version 1.2 fügt ein paar Tools bei. Daneben wurden im JDK 1.2 Tools der Versionen 1.1.x wieder beseitigt, weil sie nicht so funktioniert hatten wie geplant oder weil sie nicht mehr benötigt wurden. Standardmäßig werden die JDK-Programme in das JDK-Unterverzeichnis `\bin` installiert.

Einige allgemeine Hinweise zum JDK

- Einige der Programme des JDK lagen in bisherigen JDK-Versionen (sogar noch in 1.2-Betaversionen) jeweils in zwei Versionen vor. Es handelte sich jedes

Mal um eine Standardvariante und eine mit »_g« erweiterte, nicht optimierte Spezialversion, welche hauptsächlich für das Debugging eingesetzt werden konnte. Die Spezialversionen kommen im JDK 1.2-Final (bis auf wenige Ausnahmen) nicht mehr vor. Die jeweilige Standardversion übernimmt die Funktionalität.

- Ab dem JDK 1.2 verstehen einige Tools Non-Standard-Options. Neben dem Satz von Standardoptionen eröffnet das die Möglichkeit, einen Satz von weitergehenden Kommandozeilenoptionen zu verwenden. Diese Non-Standard-Options beginnen immer mit einem `-X`, etwa `-Xdebug` oder `-Xverify`. Wichtigster Unterschied zwischen den Standardoptionen und den Nichtstandardoptionen ist, dass die Unterstützung für die Standardoptionen im JDK 1.2 und allen zukünftigen Versionen der JVM durch Sun garantiert wird. Eine Unterstützung für die Nichtstandardoptionen wird jedoch ausdrücklich für zukünftige Versionen der JVM nicht garantiert.
- Die JDK-Version 1.1.4 ist eine Besonderheit in der JDK-1.1.x-Welt. Gerade diese Zwischenversion enthält einige Details, welche inkompatibel zum JDK 1.2 sind.
- Wenn man ein Programm des JDK ohne irgendwelche Parameter oder der Option `-Help` aufruft, bekommt man teilweise die notwendigen Angaben für die korrekte Syntax in Kurzform aufgelistet. Die JDK-Programme erweisen sich in der Handhabung leider nicht vollständig aufeinander abgestimmt. Ein Aufruf ohne Parameter oder Option schadet zwar nie, aber das Resultat ist nicht immer befriedigend.

Die Basis-Tools

Zu den Basis-Programmen des JDK 1.2-Final (also denjenigen, welche zum Erstellen von Java-Applets und -Applikationen benötigt werden) zählen die folgenden:

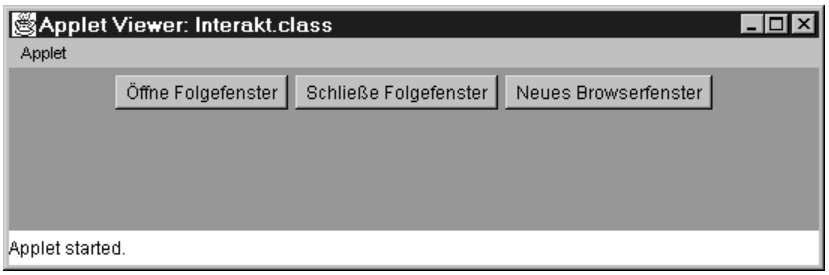
- Der `appletviewer`, um Java-Applets stand-alone zu betrachten [Solaris] [Windows].
- Der Java-Compiler `javac` zum Übersetzen von Java-Quellcode in Bytecode [Solaris] [Windows].
- Der Java-Interpreter `java` und sein nahezu identisches Schwesterprogramm `javaw`. In diesem Release gibt es nur noch diesen einen Interpreter (plus dem Interpreter für Altanwendungen `oldjava`). Er wird sowohl für die Entwicklung, als auch die reine Ausführung verwendet. Der bisherige Interpreter `jre` wird nicht mehr als Bestandteil des JDK 1.2 zur Verfügung gestellt [Solaris] [Windows].
- Der Java-Disassembler `javap` zur Rückübersetzung von Bytecode in lesbaren Klartext [Solaris] [Windows].

- `javah`, der Generator, um C-Header-Dateien und Stub-Dateien zu erstellen [Solaris] [Windows].
- Das Dokumentations-Tool `javadoc`, um aus Java-Dateien automatisch HTML-Dokumente für eine Dokumentation zu erstellen [Solaris] [Windows].
- Der Java-Debugger `jdb` [Solaris] [Windows].
- Das Java-Archive-Tool `jar` [Solaris] [Windows].
- Das Diagnose-Tool `extcheck` für Jar-File-Versionskonflikte [Solaris] [Windows].

Nachfolgend sollen die wichtigsten Basis-Tools erläutert werden.

Der Appletviewer

Der Appletviewer ermöglicht es, ohne die Hinzuziehung eines Java-fähigen Browsers ein Applet während der Laufzeit zu betrachten. Er benötigt dazu nur eine rudimentäre HTML-Datei, die eigentlich nur ein Tag mit Angabe des Applets, sowie eventuell benötigten Parametern, enthalten muss. Das Tool lädt alle Applets innerhalb des Tags und führt sie – sofern es mehrere korrespondierende Applets gibt – jeweils in einem separaten Fenster aus. Wenn eine HTML-Datei kein Tag mit Angabe eines Applets enthält, bewirkt der Aufruf des Appletviewers nichts. In der neuesten Variante versteht der Appletviewer sowohl das ursprünglich ausschließlich zu verwendende `APPLET`-Tag, als auch die Tags `OBJECT` und `EMBED` zum Einbinden.



Der Appletviewer des JDK.

Der Appletviewer hat nur wenige Aufrufoptionen. Es gibt nur die folgende, einfache Syntax:

```
appletviewer [Optionen] [URL der HTML-Datei]
```

Der Java-Compiler javac

Der Java-Compiler kompiliert die als Parameter angegebene(n) Datei(en) mit der Dateierweiterung `.java` und erzeugt daraus den Java-Bytecode in Form von Dateien mit der Erweiterung `.class`. Der Namensstamm der Datei wird für die Klassendatei übernommen. Es können sowohl Applets als auch Anwendungen mit `javac` kompiliert werden.

Es können im Prinzip beliebig viele Dateien als Parameter für die Übersetzung angegeben werden, die jedoch alle mit der Erweiterung `.java` enden müssen. `javac` produziert für jede Datei eine Bytecode-Datei mit gleichem Namen und der Erweiterung `.class`. Es ist allerdings zu beachten, dass `javac` nur eine `public`-Klasse pro Quelldatei erlaubt. Wenn innerhalb der Quelldatei mehrere Klassendefinitionen enthalten sind, dann generiert der Compiler für jede Klasse ein eigenes `.class`-File, das den Namen der Klasse erhält. Ohne entsprechende Option erzeugt `javac` das `.class`-File in dem gleichen Verzeichnis, in dem sich auch die Quelldateien befinden.

Falls innerhalb einer Quelldatei auf eine Klasse verwiesen wird, die nicht in der Kommandozeile mitangegeben wurde, so durchsucht der Compiler alle Verzeichnisse oder die Systemklassen aufgrund des neuen Suchkonzeptes. Individuell können Sie den Parameter `-classpath` angeben.

`javac`-Syntax:

```
javac [Optionen] [Dateiname] [@files]
```

Beachten Sie bitte, dass der Dateiname zwingend mit der Erweiterung `.java` eingegeben werden muss.

Neu beim aktuellen Java-Compiler ist die Möglichkeit, über das Zeichen `@` optionale Dateien anzugeben, welche selbst in jeder Zeile den Namen einer `.java`-Datei enthalten. Damit kann verhindert werden, dass die maximale Länge der Kommandozeile überschritten wird.

Die Java-Interpreter

Java stellt mehrere Kommandozeilen-Interpreter zur Verfügung. `java` und `javaw` sind identisch bis auf das kleine Detail, dass `javaw` keine dem Befehlsaufruf folgende Ausgabe von Hintergrundinformationen in einem Fenster erzeugt (die Erweiterung des Namens um den Buchstaben w deutet es an – es steht für noconsole window). Die `javaw`-Variante des Interpreters gibt es nur unter Windows. `oldjava` und `oldjavaw` unterscheiden sich untereinander genauso wie der normale Interpreter (auch die `oldjavaw`-Variante des Interpreters gibt es nur unter Windows). Deren Hauptfunktion ist die Ausführung von Applikationen, welche über einen eigenen Security-Manager verfügen, der auf dem originalen Sicherheitsmodell von Java 1.0/1.1 basiert. Dieser wird unter der Java-2-Plattform mit dem normalen Interpreter `java` eine Exception auswerfen und nicht starten. Auch sonstiger alter Java-Code kann unter Umständen damit ausgeführt werden, wenn der normale Interpreter damit nicht zurechtkommt. Allen Java-Interpretern ist gemein, dass damit Java-Anwendungen (keine Applets) ausgeführt werden. Die Syntax für die Interpreter sind weitgehend identisch:

```
java [Optionen] [Klassenname] [Argumente]
java [Optionen] -jar file.jar [Argumente]
javaw [Optionen] [Klassenname] [Argumente]
```

```
javaw [Optionen] -jar file.jar [Argumente]
oldjava [Optionen] [Klassenname] [Argumente]
oldjavaw [Optionen] [Klassenname] [Argumente]
```

`Optionen` sind diverse Übergabewerte an den Interpreter, `Klassenname` der Name der auszuführenden Klasse. Defaulteinstellung ist, dass das erste Argument, welches keine Option ist, von dem Interpreter als Name der Klasse interpretiert wird. Unbedingt zu beachten ist, dass beim Klassenname nicht nur die `.class`-Erweiterung nicht eingegeben werden muss, sondern sogar gar nicht darf. Dies ist deshalb eine häufige Fehlerquelle, weil es von der Syntaxlogik dem Compiler widerspricht, wo explizit die Erweiterung der zu verarbeitenden Datei einzugeben ist.

`Argumente` bezeichnen den oder die an die `main()`-Methode einer Java-Applikation (die Grundmethode jeder Java-Applikation) weitergegebene(n) Übergabewert(e). Jede Klasse, die in dem Aufruf durch den Klassennamen spezifiziert wird, benötigt eine solche Methode namens `main()` (`public static void main(String argv[])`). Sämtliche Argumente, die hinter dem Parameter `Klassenname` stehen, werden an diese Methode in gleicher Reihenfolge übergeben, wenn der Interpreter aufgerufen wird. Wenn mit `main()` Threads erzeugt werden, so läuft der Interpreter so lange, bis der letzte Thread beendet wird. Ansonsten endet der Interpreter nach der vollständigen Abarbeitung von `main()`.

Eventuell zur Laufzeit benötigte weitere Klassen werden vom Interpreter automatisch geladen. Die nachzuladenden Klassen müssen sich allerdings im gleichen Verzeichnis oder im Zugriffspfad befinden.

Die Tools für die erweiternden Funktionalitäten

Neben den Basis-Tools beinhaltet das JDK eine Vielzahl von weiteren Programmen, welche die zusätzlichen Funktionalitäten von Java abdecken. Die Tools für die erweiternden Funktionalitäten von Java lassen sich in verschiedene Gruppen unterteilen.

Remote Method Invocation (RMI)-Tools

Diese Programme dienen zum Generieren von Applikationen, welche über das Web oder andere Netzwerke interagieren:

- Der Java RMI Stub Converter (`rmic`) für entfernte Objekte [Solaris] [Windows]
- Java Remote Object Registry (`rmiregistry`) [Solaris] [Windows]
- Serial Version Command (`serialver`); das Tool gibt die class serialVersion UID zurück [Solaris] [Windows]
- Der RMI activation system daemon `rmid` [Solaris] [Windows]

Internationalization-Tools

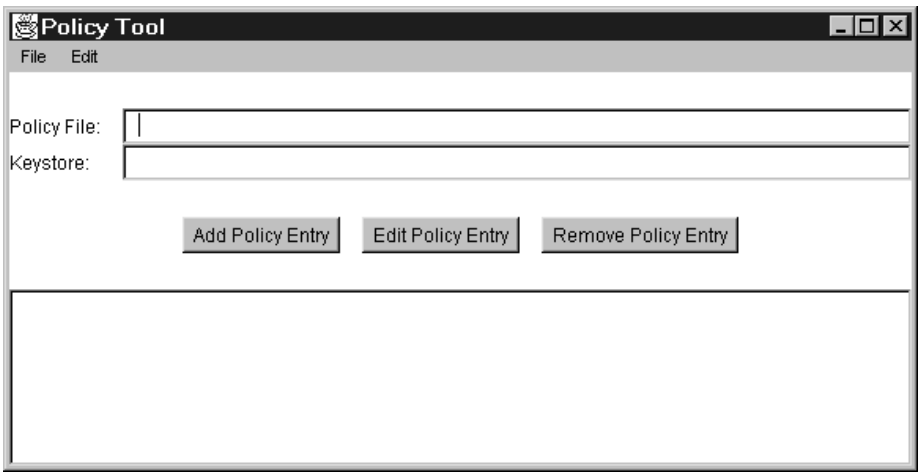
Dieses Programm dient zum Erstellen von Applikationen, welche auf landesspezifische Eigenheiten angepasst werden sollen.

- Das Programm `native2ascii` konvertiert Text in Unicode nach der Norm Latin-1 [Solaris] [Windows]

Security-Tools

Diese Programme sind zum Setzen und Verwalten von Sicherheitspolicen auf Ihrem System gedacht. Sie können damit Applikationen entwickeln, welche mit anderen Sicherheitspolicen zusammenarbeiten.

- `keytool` dient zum Schlüssel- und Zertifikatmanagement. Es wird eine Datenbank mit privaten Schlüsseln und ihren zugehörigen X.509-Zertifikaten sowie die Zertifikate von vertrauenswürdigen Entitys verwaltet. Dieses Tool basiert auf dem DSA-Algorithmus mit der SHA-1-Signatur [Solaris] [Windows].
- `jarsigner` generiert oder verifiziert eine digitale Signatur für ein JAR-File [Solaris] [Windows].
- `policytool` ist ein GUI-Tool (das bedeutet mit grafischer Oberfläche) für das Management von Policyfiles [Solaris] [Windows].



Das Policytool des JDK.

Java-IDL-Tools

Diese Programme bieten CORBA-Unterstützung für Datenbankzugriffe.

- Das unter dem JDK 1.2 neu eingeführte Programm `tnameserv` unterstützt den Zugriff auf die benannten Services.

- Neu in der Java-2-Plattform ist `idltojava`. Dieses Tool generiert `.java`-Dateien, die ein OMG-IDL-Interface mappen. Damit können in Java geschriebene Applikationen CORBA-Funktionalität nutzen. Dieses Tool ist aber nicht direkt Bestandteil des JDK, sondern kann von der Java-IDL-Webseite geladen werden. In dem Downloadfile ist eine ausführliche Dokumentation enthalten.

Eine vollständige Abhandlung (auch über weitere Tools) finden Sie in der offiziellen Dokumentation des JDK (`docs/tooldocs/tools.html`).

Die Java-Syntax

Jede Programmiersprache besteht aus einer übergeordneten Grundstruktur, welche die Syntax der Sprache bildet. Eine Art Organisation im Großen. Wir wollen in diesem Passus Java unter besagtem Aspekt behandeln. Das bedeutet, wir schauen uns Java unter folgenden Themen an:

- Datentypen
- Klassen
- Objekten
- Schnittstellen
- Ausnahmen
- Variablen
- Methoden
- Zuweisungen
- Ablaufsteuerung

und einigen mehr. Dabei soll dieses Kapitel einen grundsätzlichen Überblick über das Java-Konzept bieten. Wir verzichten deshalb an diversen Stellen auf detaillierte Syntaxerklärungen. Diese werden im Wesentlichen im Abschnitt der Java-Schlüsselworte folgen.

Die Details

Das Kapitel verfolgt folgenden Aufbau:

Kategorie	Was steht da	Wann vorhanden
Titelzeile	Der Name des Syntaxbegriffs, welcher beschrieben werden soll.	Immer vorhanden
Beschreibung	Eine Kurzbeschreibung der Bedeutung des Syntaxbegriffs.	Immer vorhanden
Anwendung	Die eigentliche Referenz. Hier erfolgt die Beschreibung der Anwendung des Syntaxbegriffs.	Immer vorhanden
Warnungen	Mögliche Gefahrenpotentiale des Syntaxbegriffs.	Optional. Dieser Abschnitt ist immer dann vorhanden, wenn eine oder mehrere Warnung(en) angebracht beziehungsweise sinnvoll ist (sind).

Kategorie	Was steht da	Wann vorhanden
Beispiele	Kommentierte Beispiele, welche die Anwendung des Syntaxbegriffs erläutern.	Immer vorhanden

Hinweis: Der in verwandten Titeln der Buchreihe besprochene Abschnitt über Parameter, die für den Befehl sinnvoll oder notwendig sind, macht bei einem Aufbau dieses Kapitels über die Java-Syntax (und über die gesamte Java-Referenz) keinen Sinn. Wir behandeln hier keine Befehle im üblichen Sinn, sondern übergeordnete Strukturen. Entsprechend werden Sie in diesem Kapitel auch keine Rubrik »Verwandte Befehle« finden. Diese Rubrik werden Sie jedoch im Kapitel über Java-Schlüsselworte finden.

Anweisungen

Anweisungen sind sprachspezifische Befehle, welche Java – wie nahezu jede Programmiersprache – in zahlreichen Varianten kennt.

Beschreibung

Anweisungen werden in Java der Reihe nach oder aufgrund eines bestimmten Effektes ausgeführt.

Anwendung

Java kennt zahlreiche Arten von Anweisungen. Man unterscheidet sie wie folgt:

Anweisungsart	Beschreibung
Leere Anweisungen	Keine Funktion, nur Platzhalter.
Blockanweisungen	Ein Zusammenfassen von größeren Quellcodeabschnitten zu Blockstrukturen. Dies erfolgt in Java immer mittels geschweifter Klammern.
Bezeichnete Anweisungen	Jede Anweisung kann mit einer Bezeichnung beginnen. Diese Bezeichnungen dürfen keine Java-Schlüsselworte, bereits festgelegte lokale Variablen oder sonst schon in diesem Modul verwendeten Bezeichnungen sein. Der Bezeichnung folgt bei einer solchen bezeichneten Anweisung immer ein Doppelpunkt.
Deklarationen	Die Einführung eines primitiven Datentyps, eines Feldes, einer Methode, einer Klasse, einer Schnittstelle oder eines beliebigen Objektes.
Ausdrucksanweisungen	Ausdrucksanweisungen bewirken eine Wertveränderung oder -ausgabe. Alle Ausdrucksanweisungen müssen in Java mit einem Semikolon beendet werden und werden immer vollständig durchgeführt, bevor die nächste Anweisung ausgeführt wird. Java verwendet sieben verschiedene Ausdrucksanweisungsarten: • Zuordnung • Pre-Inkrement • Pre-Dekrement • Post-Inkrement • Post-Dekrement • Methodenaufruf • Zuweisungsausdruck Eine Zuweisungsanweisung kann einen Ausdruck rechts von dem Gleichheitszeichen (dem Zuweisungsoperator =) stehen haben. Dieser Ausdruck kann jeder der sieben Ausdrucksanweisungen sein. Es darf in Java immer nur die rechte Seite einer solchen Zuweisung festgelegt werden.

Anweisungsart	Beschreibung
Auswahlanweisungen	Auswahlanweisungen suchen einen von mehreren möglichen Kontrollflüssen aus. Es gibt in Java drei verschiedene Arten von Auswahlanweisungen: • `if`-Auswahlanweisung • `if-else`-Auswahlanweisung • `switch-case-default`-Auswahlanweisung
Iterationsanweisungen	Iterationsanweisungen sind eine Angabe, unter welchen Voraussetzungen und wie oft nachfolgende Anweisungen ausgeführt werden. Es gibt in Java drei Arten von Iterationsanweisungen: • `while`-Iterationsanweisung • `do`-Iterationsanweisung • `for`-Iterationsanweisung
Sprunganweisungen	Sprunganweisungen geben die Steuerung entweder an den Anfang oder das Ende des derzeitigen Blocks, oder aber an bezeichnete Anweisungen weiter. Es gibt in Java vier Arten von Sprunganweisungen: • `break`-Sprunganweisungen • `continue`-Sprunganweisungen • `return`-Sprunganweisungen • `throw`-Sprunganweisungen
Synchronisationsanweisungen	Eine Anweisungsform für den Umgang mit Multithreading. Damit werden Methoden oder Blöcke markiert, die eventuell vor gleichzeitiger Verwendung geschützt werden sollen. Es gibt in Java die folgende Synchronisationsanweisung : • `synchronized`-Synchronisationsanweisung
Schutzanweisungen	Java verfolgt zum Abfangen von Laufzeitfehlern ein Konzept, welches mit sogenannten Ausnahmen arbeitet. Bei Ausnahmen handelt es sich um nicht planbare Situationen, welche eine unmittelbare Reaktion durch das Programm beziehungsweise den Anwender notwendig machen. Schutzanweisungen werden zur sicheren Handhabung von solchem Code, der Ausnahmen auslösen könnte, verwendet. Es gibt die Schutzanweisungen über die Kombination der Schlüsselworte `try`, `catch` und `finally`.
Unerreichbare Anweisungen	Sie erzeugen einen Fehler zur Kompilierzeit.

Beispiele

Eine Block-Anweisung:

```
{
...
}
```

Eine bezeichnete Anweisung:

```
Ende:
```

Eine Deklaration:

```
int i;
```

Zuordnung-Ausdrucksanweisungen:

```
a = 42
```

Pre-Inkrement-Ausdrucksanweisungen:

```
++wert
```

Pre-Dekrement-Ausdrucksanweisungen:

```
--wert
```

Post-Inkrement-Ausdrucksanweisungen:

```
wert++
```

Post-Dekrement-Ausdrucksanweisungen:

```
wert--
```

Methodenaufruf-Ausdrucksanweisungen:

```
System.out.println("Hello World!")
```

Zuweisungsausdruck-Ausdrucksanweisungen:

```
MeineKlasse wert = new MeineKlasse();
```

Eine Auswahlanweisung:

```
if (test < 4)
{
...
}
```

Arrays

Arrays sind eine Ansammlung von Objekten oder primitiven Datentypen, die über einen gemeinsamen Namen angesprochen werden können.

Beschreibung

Felder beziehungsweise Datenfelder (engl. Arrays) gehören neben Klassen und Schnittstellen zu den Referenzvariablen in Java. Sie sind in Java immer aus Objekten beziehungsweise primitiven Datentypen zusammengesetzt und können über einen gemeinsamen Namen und einen Index angesprochen werden.

Anwendung

Ein Array ist eine Ansammlung von Objekten eines bestimmten Typs beziehungsweise primitiven Datentypen (es sind keine verschiedenen Typen innerhalb eines Arrays erlaubt), welche über einen laufenden Index adressierbar sind. Arrays sind damit nichts anderes als (besondere) Objekte. Sie werden wie normale Objekte dynamisch von Java erzeugt und am Ende ihrer Verwendung vom Garbage Collector (der automatischen Speicherbereinigung von Java) beseitigt.

Weiterhin stehen in Arrays als Ableitung der obersten Klasse `Object` alle Methoden dieser Klasse zur Verfügung. Arrays können aus sämtlichen primitiven Variablentypen aufgebaut sein, aber auch aus beliebigen Objekten. Insbesondere können diese Objekte wieder selbst Felder sein, womit Java ein Array von Arrays erzeugt. Dies ist auch die einzige Möglichkeit für multidimensionale (gelegentlich auch mehrdimensionale genannt) Arrays, da sie von Java nicht direkt unterstützt werden (anders als beispielsweise in Pascal oder auch der eng verwandten Sprache C/C++). Da jedoch im Prinzip beliebig viele Ebenen von Datenfeldern ineinander verschachtelt werden können, lässt sich so ein nicht unterscheidbares Analogon zu multidimensionalen Feldern beliebiger Dimension schaffen.

Gegenüber normalen Objekten haben Arrays zwei wesentliche Einschränkungen:

- Arrays haben keine Konstruktoren. Statt dessen wird der `new`-Operator mit spezieller Syntax aufgerufen.
- Es können keine Subklassen eines Arrays definiert werden.

Um ein Datenfeld in Java zu erstellen, muss man drei Schritte durchführen:

1. Das Deklarieren des Datenfeldes
2. Das Zuweisen von Speicherplatz
3. Das Füllen des Datenfeldes

Es ist möglich, mehrere der Schritte zur Erstellung eines Feldes mit einer Anweisung zu erledigen und ein Datenfeld kann bei der Initialisierung sowohl ganz als auch nur teilweise gefüllt werden. Besonders beachtet werden sollte, dass die Indizierung von Feldern mit 0 beginnt. Dabei müssen Feldindizes positiv und vom Typ

`int` sein und dies beschränkt die größtmögliche Feldgröße auf 2.147.483.647. beginnt. Dabei müssen Feldindizes positiv und vom Typ `int` sein und dies beschränkt die größtmögliche Feldgröße auf 2.147.483.647.

Ein Feld ist in Java immer eine Variable, kein (ein Verweis auf eine konkrete Adresse im Speicher), wie es beispielsweise in C/C++ der Fall ist. Es kann auf kein Feldelement in Java zugegriffen werden, das noch nicht erstellt worden ist. Wenn ein Zugriff auf Elemente außerhalb der Array-Grenzen erfolgt, wird entweder ein Kompilierungsfehler (sofern dieser Zugriff bereits im Sourcecode ersichtlich ist) oder ein Laufzeitfehler erzeugt (bei dynamisch erzeugten oder veränderten Feldern). Dadurch wird das Programm vor einem Absturz bewahrt.

Für das konkrete Deklarieren von Feldern in Java wird eine Variable, genauer gesagt eine Referenz, erstellt, die auf den Speicherbereich verweisen wird, wo die eigentlichen Daten abgelegt werden. Dabei müssen der Datentyp und der Name der Feld-Variablen immer festgelegt werden, nicht jedoch bereits die Größe. Im Unterschied zu normalen Variablen muss mit eckigen Klammern die Variable als Feld gekennzeichnet werden. Diese eckigen Klammern können nach dem Datentyp oder nach dem Namen der Variablen notiert werden. Java unterstützt beide Syntaxtechniken. Sofern innerhalb der Klammern eine Zahl angegeben wird, bekommt das Array bereits bei der Deklaration eine feste Größe, ansonsten muss die Größe später festgelegt werden. Arrays mit anderen Arrays als Inhalt werden deklariert, indem pro Dimension ein weiteres Paar an eckigen Klammern angefügt wird.

Das Erstellen von Feld-Objekten erfolgt entweder mittels des `new`-Operators (direkte Erzeugung eines Feld-Objektes) oder durch direktes Initialisieren des Array-Inhaltes. Für die direkte Technik der Erzeugung eines Feld-Objektes müssen nach dem Gleichzeichen die Elemente des Arrays in geschweiften Klammern und mit Komma getrennt angegeben werden. Beim Erzeugen eines Feld-Objektes mit `new` werden alle Elemente des Arrays automatisch initialisiert. Dabei gelten bei Feldern von primitiven Datentypen die Defaultwerte des jeweiligen Datentyps.

Das Zugreifen auf Feldelemente erfolgt über den Namen der Feldvariablen und dem Index in eckigen Klammern. Bei Feldern mit Feldern als Inhalt erfolgt der Zugriff, indem die eckigen Klammern pro Dimension nacheinander angegeben werden müssen.

Warnungen

Felder sind in Java gegenüber anderen Programmiersprachen wie C/C++ oder PASCAL anders konzipiert. Zum einen stellt Java sicher, dass keine fehlgeleiteten Zugriffe erfolgen. Zum anderen erfolgt der Zugriff über die aufeinanderfolgende Notation der eckigen Klammern (pro Dimension). Ein Zugriff auf das dritte Element in der dritten Spalte erfolgt beispielsweise über `meinArray [2][2];`.

Der Unterschied gegenüber Arrays in vielen anderen Programmiersprachen ist, dass die eckigen Klammern in der beschriebenen Weise angegeben werden müssen. Ein Zugriff in der Form `meinArray [2,2]` oder ähnlich wird einen Fehler erzeugen.

Die Indizierung eines Arrays beginnt immer mit 0.

Beispiele

Deklaration von Feldern:

```
double meinArray[];
double[] meinArray;
```

Deklaration von einem dreidimensionalen Feld ohne Angabe der Größe:

```
int meinArray [] [] [];
```

Erzeugung eines Feldes mit dem `new`-Operator

```
double[] meinArray = new double[42];
```

Erzeugung eines Feldes mit direktem Initialisieren des Array-Inhaltes:

```
int[] meinArray = {1,2,3,4,5,9,1,22,44,42,12,14};
```

Zugriff auf das dritte Element eines Feldes:

```
meinArray [2];
```

Ein Feld mit einem Feld als Inhalt (2x2-Array):

```
int[][] meinArray = {{1,2},{3,1}};
```

Ausdrücke

Das Ergebnis einer Sprachoperation.

Beschreibung

Ausdrücke sind das Ergebnis von der Verbindung von Operanden und Operatoren gemäß den syntaktischen Regeln von Java.

Anwendung

Ausdrücke werden für die Durchführung von Operationen (Manipulationen) an Variablen oder Werten verwendet. Dabei sind Spezialfälle wie arithmetische Konstanten kein Widerspruch, sondern nur die leere Operation. Ein Typ von Ausdrükken in Java drückt einen Wert entweder direkt oder durch Berechnung aus. Ein anderer Typ – der Kontrollfluss-Ausdruck – legt den Ablauf von Programmausführungen fest.

Ausdrücke können Konstanten, Variablen, Schlüsselworte, Operatoren und andere Ausdrücke beinhalten.

Beispiele

Arithmetische Konstante:

```
42
```

Multiplikativer Ausdruck:

```
3*4
```

Additiver Ausdruck:

```
1 + 3
```

Bitverknüpfungs-Ausdruck:

```
2^32
```

Zuweisungs-Ausdruck:

```
x=42
```

Casting

Die Konvertierung von Datentypen, Objekten oder Schnittstellen in Java.

Beschreibung

Unter Casting versteht man die Umwandlung von einem Datentyp in einen anderen Datentyp oder die Umwandlung von einer Klasseninstanz in ein anderes Objekt beziehungsweise eine Schnittstelle.

Anwendung

Java ist eine streng typisierte Sprache, weil sehr intensive Typüberprüfungen stattfinden. Daher gelten strikte Beschränkungen für die Umwandlung (Konvertierung) von Werten eines Datentyps in einen anderen. Java unterstützt zwei unterschiedliche Arten von Konvertierungen:

- Ad-hoc-Konvertierungen
- Explizite Konvertierungen

Bei Ad-hoc-Typkonvertierungen werden durch Java bei der Bewertung von Ausdrücken notwendige Typkonvertierungen automatisch durchgeführt. Dies geschieht aber nur innerhalb sehr enger Grenzen. Insbesondere muss der entstehende Datentyp des darzustellenden Wertebereiches größer oder gleich dem zu konvertierenden Datentyp sein. Dabei ist auch die Darstellung als Zweierkomplement[3] mit Vorzeichen oder nicht (etwa der Zeichentyp) zu beachten. Es ist also nicht nur die reine Bitlänge des beteiligten Datentyps entscheidend. Java stellt damit sicher, dass bei Ad-hoc-Typkonvertierungen keine Informationen verloren gehen können. Ansonsten gelten folgende Regeln:

- Bei Operationen mit ausschließlich ganzzahligen Operanden wird, wenn einer der beiden Operanden den Datentyp `long` hat, der andere gleichfalls zu `long` konvertiert; ansonsten werden beide Operanden zu `int` konvertiert. Das Ergebnis ist vom passenden Typ. Nur wenn der ausgegebene Wert zu groß ist, um im Wertebereich von `int` dargestellt zu werden, wird der Typ `long` verwendet.

3. **Ganzzahlige Variablen werden in Java allesamt als Zweierkomplementzahlen mit Vorzeichen verwendet. Unter einem Zweierkomplement ist eine Methode zu verstehen, um negative ganze Zahlen binär darzustellen. Dabei wird bei der Darstellung einer negativen Zahl von der analogen positiven Zahl (dem positiven Komplement) die Zahl 1 abgezogen und dann sämtliche Bits umgedreht. Die Null zählt noch zum positiven Anteil. In Java hat dies die zwangsläufige Folge, daß das am weitesten links stehende Bit als Vorzeichenbit dient. Wenn das Vorzeichenbit den Wert 1 hat, dann ist der Wert negativ, sonst positiv.**

- Bei Operationen mit wenigstens einem Fließkomma-Operanden wird, wenn einer der Operanden den Datentyp `double` hat, der andere ebenso zu `double` konvertiert, und das Ergebnis ist dann ebenfalls vom Typ `double`; sonst werden beide Operanden zum Datentyp `float` konvertiert. Das Ergebnis des Ausdrucks ist ebenfalls vom Typ `float`.
- Bei der Verknüpfung von einer Zeichenkette und einem Wert (mit dem Verknüpfungsoperator `+`), der keine Zeichenkette ist, wird dieser vor dem Verbinden automatisch zu einer Zeichenkette konvertiert.

Bei expliziten Konvertierungen löst der Programmierer selbst eine Umwandlung in einen anderen Datentyp oder ein anderes Objekt aus. Dieses Casting erfolgt mittels eines Casting-Operators (manchmal Festlegungsoperator genannt). Der Casting-Operator besteht nur aus einem Datentypnamen beziehungsweise dem Typ des Zielobjektes in runden Klammern. Er ist ein einstelliger Operator mit hoher Priorität und steht vor seinem Operanden.

Ein Casting-Operator ist also bei der Konvertierung von Datentypen immer von der folgenden Form:

```
(Datentyp) Wert
```

Es gibt im Java-Standard für jeden Datentyp außer `boolean` einen Casting-Operator.

Die Typkonvertierung ist aber wie gesagt nicht nur auf primitive Datentypen beschränkt. Mit Einschränkungen lassen sich sogar Klasseninstanzen in Instanzen anderer Klassen konvertieren. Die wesentliche Einschränkung ist, dass die Klassen durch Vererbung miteinander verbunden sein müssen. Allgemein gilt, dass ein Objekt einer Klasse auf seine Superklasse gecastet werden kann. Bezüglich der Subklasse (welche ja in der Regel durch Erweiterungen mehr Informationen enthält) gibt es im Allgemeinen Probleme. Beim Konvertieren von einer Klasse auf eine Instanz seiner Superklasse gehen die spezifischen Informationen, welche nur in der zu konvertierenden Subklasse vorhanden sind, natürlich verloren. Die Syntax zum Konvertieren ist analog der Fall des Castings bei primitiven Datentypen:

```
(Klassenname) Objekt
```

Der Klassenname ist der Name der Klasse, in die das Objekt konvertiert werden soll. `[Objekt]` ist eine Referenz auf das konvertierte Objekt. Casting erstellt eine neue Instanz der neuen Klasse. Das alte Objekt existiert unverändert weiter.

Mit Einschränkungen lassen sich auch Klasseninstanzen in Schnittstellen konvertieren. Dabei ist allerdings zwingend, dass die Klasse selbst oder eine Superklasse des Objektes die Schnittstelle implementiert. Durch Casting eines Objektes in eine Schnittstelle kann dann eine Methode dieser Schnittstelle verwendet werden, obwohl die Klasse des Objektes diese Schnittstelle unter Umständen nicht direkt implementiert hat.

Dagegen ist die Konvertierung von primitiven Datentypen in Objekte und umgekehrt nicht möglich! Weder ad hoc noch durch explizites Casting. Statt dessen gibt es im Java-Paket `java.lang` als Ersatz dafür Sonderklassen, welche primitiven Datentypen entsprechen. Mit den in den Klassen definierten Klassenmethoden kann mit Hilfe des `new`-Operators jeweils ein Gegenstück zu jedem primitiven Datentypen erstellt werden. Die Konvertierung in umgekehrte Richtung funktioniert ebenfalls. Dazu gibt es in den jeweiligen Klassen passende Methoden zum Extrahieren von primitiven Datentypen.

Warnungen

- Boolesche Variablen sind dem eigenen Java-Typ `boolean` zugeordnet und können nur die Werte `true` oder `false` annehmen. Die Zuordnung eines booleschen Datentyps zu einer Zahl erzeugt einen Fehler durch den Compiler. Auch Zahlenoperationen können mit diesem Typ explizit nicht durchgeführt werden. Werte vom Typ `boolean` sind zu allen anderen primitiven Datentypen inkompatibel und lassen sich insbesondere im Javastandard nicht durch Casting in andere Typen überführen[4].
- Casting hat eine höhere Priorität als Arithmetik. Deshalb müssen arithmetische Operationen in Verbindung mit Casting in Klammern gesetzt werden.

Beispiele

Casting in einen `byte`-Datentyp

```
(byte) (4.0/2.0)
```

Casting in einen `short`-Datentyp

```
(short) x
```

Casting in einen `int`-Datentyp

```
(int) (x/y)
```

Konvertierung von einem primitiven Datentyp in ein Objekt, welches das Gegenstück zu dem primitiven Datentyp darstellt:

```
Integer meinObjekt = new Integer(42);
```

Extrahieren von dem in dem Objekt gespeicherten Wert und Zuweisen zu einem primitiven Datentyp mit einer geeigneten Klassenmethode:

```
int meineVariable = meinObjekt.intValue();
```

4. Allerdings lässt sich leicht ein Casting-Operator erstellen, welcher eine solche Funktionalität bietet.

Datentypen

Die Angabe des Typs eines einfachen Objektes.

Beschreibung

Ein Datentyp ist die Angabe, wie ein einfaches Objekt (beispielsweise eine Variable) im Rahmen einer Computersprache im Hauptspeicher eines Computers dargestellt wird. Eine solche Angabe enthält normalerweise ebenfalls Hinweise darüber, welche Operationen bezüglich des einfachen Objektes erlaubt sind. Jede Programmiersprache bietet »fest eingebaute« Datentypen (sog. einfache oder primitive Datentypen). In Java kann man darüber hinaus als Programmierer eigene komplexe Datentypen definieren mit Hilfe des Klassenkonzeptes.

Anwendung

In Java sind sämtliche primitiven Datentypen plattformunabhängig, das bedeutet, sie sind sowohl bezüglich ihrer Länge als auch bezüglich ihrer erlaubten Arithmetik auf jeder Java unterstützenden Plattform eindeutig festgelegt und verhalten sich überall gleich. Des Weiteren haben sie immer einen wohldefinierten Default-Anfangswert (Ausnahme – der Einsatz als lokale Variablen). Java besitzt acht primitive Datentypen:

- Vier Ganzzahltypen mit Vorzeichen zum Darstellen von Ganzzahlwerten mit unterschiedlichen Wertebereichen per Zweierkomplement.
- Einen logischen Datentyp, der nur zwei boolesche Werte annehmen kann und keine Zuordnung zu einer Zahl oder Zahlenoperationen gestattet. Der logische Datentyp in Java ist explizit zu allen anderen primitiven Datentypen inkompatibel und lässt sich auch nicht durch Casting in andere Typen überführen.
- Zwei Gleitzahltypen mit Vorzeichen zur Darstellung von Gleitkommazahlwerten mit unterschiedlichen Wertebereichen nach der internationalen Norm IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std. 754-1985 (IEEE, New York) zur Definition von Gleitpunktzahlen und Arithmetik. Beide Typen besitzen ein Literal zur Darstellung von plus/minus Unendlich sowie den Wert `NaN` (Not a Number) zur Darstellung von nichtdefinierten Ergebnissen.
- Einen Zeichentyp zur Darstellung eines Zeichens des Unicode-Zeichensatzes. Dabei wird bei der Darstellung von alphanumerischen ASCII-Zeichen dieselbe Kodierung wie beim ASCII-Zeichensatz verwendet, aber das höchste Byte ist auf 0 gesetzt. Der Datentyp ist vorzeichenlos!

Die Ganzzahltypen in Java

Bezeichnung	Länge	Defaultwert	Wertebereich
`byte`	8 Bit	`0`	Kleinster Wertebereich von `-128` bis `+127`
`short`	16 Bit	`0`	Kurze Darstellung von `-32.768` bis `+32.767`
`int`	32 Bit	`0`	Standardwertebereich von `-2.147.483.648` bis `+2.147.483.647`
`long`	64 Bit	`0`	Größter Wertebereich von `-9.223.372.036.854.775.808` bis `+9.223.372.036.854.775.807`

Die Gleitzahltypen in Java

Bezeichnung	Länge	Defaultwert	Wertebereich
`double`	64 Bit	`0.0`	Größter Wertebereich mit Werten zwischen ca. `+/- 1,8E+308`
`float`	32 Bit	`0.0`	Kürzester Wertebereich mit Werten zwischen ca. `+/- 3,4E+38`

Der Zeichentyp in Java

Bezeichnung	Länge	Defaultwert	Wertebereich
`char`	16 Bit	`\u0000`	Wertebereich von `\u0000` bis `\uFFFF`

Der logische Datentyp in Java

Bezeichnung	Länge	Defaultwert	Wertebereich
`boolean`	1 Bit	`false`	Wertebereich `true` oder `false`

Mit den Datentypen in Java lassen sich die üblichen Operationen (Zuweisungen, Überprüfung auf Gleichheit, Ungleichheit oder Relation zueinander, binäre und logische Arithmetik etc.) ausführen, sofern sie für den betreffenden Datentyp Sinn machen. Diese sind für die unterschiedlichen Datentypen immer dann identisch oder ähnlich, wenn dies logisch gesehen sinnvoll ist. In den meisten Fällen handelt es sich bei Operation auf größeren (sowohl im Sinne der Bits, als auch der Funktionalität) Datentypen um eine natürliche Erweiterung der Operationen, die mit kleineren Typen durchgeführt werden. So sind die meisten Fließkommaoperationen und Ganzzahloperationen identisch. Wichtigste Ausnahmen sind diejenigen Fließkommaoperationen, welche den Nachkommateil betreffen und bei Ganzzahloperationen keinen Sinn machen. Diese müssen für Fließkommaoperationen individuell definiert werden. Aber auch umgekehrt kann es vorkommen, dass einzelne Operationen auf größeren Datentypen nicht sinnvoll sind, obwohl sie auf kleineren Datentypen sinnvoll sind (etwa die auf ganzen Zahlen definierte binäre Verschiebung, die auf Gleitzahlen nicht sinnvoll ist).

Warnungen

- Die Zuordnung eines booleschen Datentyps mit einer Zahl oder eine Zahlenoperation damit erzeugt einen Fehler durch den Compiler.
- Java erzeugt keinerlei Ausnahmen bei arithmetischen Operationen. Ein Overflow (das bedeutet ein größeres Ergebnis durch eine Operation, als durch den Wertebereich des jeweiligen Zieltyps ausgedrückt werden kann) oder die Division einer Zahl (ungleich Null) durch Null und ähnliche Operationen erzeugen die Ausgabe von positiven beziehungsweise negativen unendlichen Werten. Dies sind sinnvolle Werte, die mit Vergleichsoperatoren ausgewertet werden können. Ein Unterlauf (das heisst ein kleineres Ergebnis – außer Null – durch eine Operation, als durch den Wertebereich des jeweiligen Zieltyps ausgedrückt werden kann) gibt einen speziellen Wert aus, der positiv oder negativ Null genannt wird. Das Teilen von Null durch Null ergibt den Wert `NaN` (keine Zahl). Auch dies ist durchaus ein sinnvoller Wert, denn er kann mit Vergleichsoperatoren ausgewertet werden.

Beispiele:

Eine Methodendeklaration mit einem booleschen Rückgabewert:

```
boolean jaodernein();
```

Eine Methodendeklaration mit einem Rückgabewert vom Typ `byte`:

```
byte test();
```

Eine Methodendeklaration mit einem Rückgabewert vom Typ `char`:

```
char test();
```

Eine Methodendeklaration mit einem Rückgabewert vom Typ `double`:

```
double test();
```

Eine Methodendeklaration mit einem Rückgabewert vom Typ `int`:

```
int test();
```

Eine Methodendeklaration mit einem Rückgabewert vom Typ `long`:

```
long test();
```

Eine Methodendeklaration mit einem Rückgabewert vom Typ `short`:

```
short test();
```

Eine boolesche Variable:

```
boolean test;
```

Eine Variable vom Typ `byte`:

```
byte test;
```

Eine Variable vom Typ `char`:

```
char test;
```

Eine Variable vom Typ `float` mit einer Zuweisung:

```
float gleitzahl = 0.123;
```

Eine Variable vom Typ `int`:

```
int test;
```

Eine Variable vom Typ `long`:

```
long test;
```

Eine Variable vom Typ `short`:

```
short test;
```

Klassen

Die Zusammenfassung von Beschreibungen zur Erstellung von Objekten.

Beschreibung

Im Rahmen der objektorientierten Programmierung werden ähnliche Objekte zusammengefasst, um eine leichtere Klassifizierung von diesen Objekten zu ermöglichen. Die Eigenschaften der Objekte werden in übergeordneten Gruppierungen – den Klassen – als eine Gruppe zusammenhängender Informationen und Methoden gesammelt und für eine spätere Erzeugung von realen Objekten verwendet.

Anwendung

Java ist streng aus Klassen aufgebaut, die allesamt Ableitungen einer einzigen obersten Klasse – der Klasse `Object` – sind. Um das objektorientierte System konsequent einzuhalten, ist auch diese oberste Klasse Instanz einer Metaklasse, deren einzige Objekte Klassen sein können. Jede Klasse definiert das Verhalten eines daraus erzeugten Objektes (der Instanz) durch spezifische Methoden und Variablen.

Klassen werden in Java hierarchisch aufgebaut – von allgemein bis fein. Diese Hierarchie steht über den OO-Mechanismus der Vererbung in Beziehung zueinander, wobei Java nur Einfachvererbung (keine Mehrfachvererbung wie beispielsweise C/C++) erlaubt. Gemeinsame Erscheinungsbilder werden in der objektorientierten Philosophie möglichst in einer maximal allgemeinen Klasse zusammengefasst. Erst wenn Unterscheidungen möglich beziehungsweise notwendig sind (Eigenschaften oder Funktionalitäten, welche nicht für alle Mitglieder einer Klasse gelten können), werden Untergruppierungen – untergeordnete Klassen – gebildet. Jede untergeordnete Klasse bekommt sämtliche (nichtgeheimen) Eigenschaften der übergeordneten Klasse vererbt und sollte diese sinnvoller Weise mindestens um eine neue Eigenschaft erweitern (sonst ist sie ja identisch). Eine solche Erweiterung kann etwa daraus bestehen, dass Attribute hinzugefügt, neue Methoden spezifiziert, vorhandene Methoden überladen oder geerbte Methoden ergänzt werden. Eine übergeordnete Klasse wird Superklasse oder Basisklasse, eine untergeordnete Klasse die Subklasse genannt. Die ineinander geschachtelten Klassen bilden einen sogenannten Klassenbaum. Dieser kann in Java im Prinzip beliebig tief werden. Abgeleitete Klassen übernehmen die Eigenschaften und Methoden aller übergeordneten Klassen, wobei Übernehmen nicht heisst, dass eine Subklasse die Befehle und Felder der Superklasse in ihre eigene Deklaration kopiert. Statt dessen gibt es nur eine formale Verknüpfung zwischen den Klassen (analog dem Verwenden von Bibliotheken).

Das Manipulieren der Eigenschaften eines Objekts erfolgt bei Java wie in den meisten anderen objektorientierten Programmiersprachen über die Punkt-Notierung (DOT-Notation). Um den Wert einer Eigenschaft anzusprechen, wird das Objekt angegeben, dann folgt ein Punkt und anschließend die Eigenschaft. Diese Form der

Auswahl von Elementen in einer Klassenhierarchie bewirkt einen sogenannten Nachrichtenselektor. Konkret ist dies etwa der Methodenname (einer Botschaft). Ist der Empfänger der Botschaft nun in der aktuellen Klasse nicht vorhanden, so wird die gewünschte Methode/Variable in der nächsthöheren Superklasse gesucht. Ist das Ziel dort desgleichen nicht vorhanden, erfolgt die Suche in der nächsthöheren Klasse, bis die oberste Stufe des Klassenbaums erreicht ist (bei Java immer die Klasse `Object`). Die Ausführung einer Methode oder der Zugriff auf eine Variable erfolgt in der ersten Klasse, in der sie gefunden wird (von der aktuellen Klasse in der Hierarchie aufwärts gesehen). Gibt es im Klassenbaum keine Methode/Variable eines angegebenen Namens, so kommt es zu einer Fehlermeldung. Klassen auf derselben Ebene oder in anderen Zweigen werden in Java nicht durchsucht.

Die allgemeine Struktur einer Java-Klasse besteht aus der Deklaration und dem Body (Körper). Die Deklaration beginnt mit dem Schlüsselwort `class`, gefolgt von dem Namen der Klasse. Der Klassenname muss die allgemeinen Regeln für Token unter Java befolgen. Danach folgt innerhalb der geschweiften Klammern der Klassenkörper. Bei der Deklaration einer Klasse werden zusätzlich (optional) Modifier, eine Superklasse und eventuelle Schnittstellen angegeben. Superklassen werden bei einer Klasse in Java über das Schlüsselwort `extends` deklariert, Schnittstellen mittels des Schlüsselwortes `implements`. Dabei können Klassen genau eine Superklasse haben (und haben auch eine, selbst wenn diese nicht explizit deklariert wird – dann gilt die Defaultsuperklasse), aber eine beliebige Anzahl von Schnittstellen.

Wenn explizit vorhanden, eröffnen Modifier eine Klassendeklaration und legen fest, wie die Klasse bezüglich der Sichtbarkeit der Klasse (etwa `public`) und dem Benutzungsgrad gehandhabt (`final` und `abstract`) werden kann.

Klassen haben immer einen voreingestellten Defaultstatus – freundlich. Dies bedeutet, dass diese Klassen erweitert und von anderen Klassen benutzt werden können, aber nur von Objekten innerhalb desselben Pakets. Eine als `public` deklarierte Klasse bedeutet, dass alle Objekte auf die Klasse zugreifen können (auch Objekte, welche nicht zum Paket dieser Klasse gehören). Andere Modifier wie `protected` machen bei Klassen kaum Sinn und unterbleiben. Über den Benutzungsgrad kann festgelegt werden, ob Klassen abgeleitet werden können oder nicht. Der Modifier `final` legt fest, dass eine Klasse nicht weiter abgeleitet werden kann. Unter einer abstrakten Klasse versteht man eine Klasse, von der nie eine direkte Instanz benötigt wird und von welcher auch keine Instanz gebildet werden kann. Prinzipiell dienen abstrakte Klassen dazu, unvollständigen Code zu deklarieren.

Jede Klasse in Java verfügt über spezielle Methoden (mindestens eine, aber oft auch mehrere), welche bei der konkreten Erstellung einer Instanz genutzt werden. Es handelt sich dabei um sogenannte Konstruktoren. Sie dienen dazu, ein Objekt zu initialisieren und bestimmte Eigenschaften der Instanz festzulegen, Aufgaben auszuführen und Speicher für die Instanz zu allokieren. Konstruktoren müssen immer den gleichen Namen wie die Klasse selbst haben! Weiterhin dürfen Konstruktoren

(obwohl sie Methoden sind) keine Rückgabeparameter (nicht einmal die Deklaration als `void` ist erlaubt) haben. Sie geben ja die Instanz der Klasse zurück.

Ein Konstruktor wird bei jedem Erstellen eines Objektes aufgerufen. Etwa, wenn als Rückgabewert einer Methode ein Objekt erzeugt wird oder vor allem bei der expliziten Erstellung einer Instanz mittels des `new`-Operators mit der folgenden Syntax:

```
instanzKlasse = new KlassenName(parameter);
```

Dabei ist `instanzKlasse` die Variable, welche die Instanz der Klasse `KlassenName` aufnimmt und die Angabe `parameter` bezeichnet optionale Parameter. Die Gesamtangabe `KlassenName(parameter)` ist die Konstruktormethode. Anzahl und Typ von den Parametern legt (bei mehreren Konstruktoren) die konkret zu verwendende Konstruktormethode einer spezifischen Klasse fest. Wenn in einer Klasse keine explizite Konstruktormethode definiert wurde, dann ruft Java eine Default-Konstruktormethode ohne Parameter auf.

Unter der Javaversion 1.1 wurden die sogenannten inneren Klassen (Inner Classes) eingeführt, mit denen Java eine Blockstruktur innerhalb einer Klasse (auf Klassenebene) unterstützt. Klassen und Schnittstellen können über dieses Konzept innerhalb anderer Klassen eingebettet werden. Sie können von dort aus ausschließlich die Klassen unterstützen, in welche sie integriert sind. Der vollständige Name einer inneren Klasse wird durch die umschließende Klasse bestimmt, dass heisst über den Namen der äußeren Klasse, einen Punkt und den Namen der inneren Klasse. Innere Klassen können dieselben Zugriffsmodifizierer verwenden wie die anderen Mitglieder einer Klasse.

Anonymous Classes steht für eine Abart der inneren Klassen. Es handelt sich um eine Kurzform von inneren Klassen. Sie haben allerdings keinen Namen, nur eine Implementation mit `new`. Der Compiler generiert bei den anonymen Klassen eine namenlose (anonymous) Klasse, welche wie spezifiziert eine bestehende Klasse dann überschreibt. Das Ende einer anonymen Klasse wird durch das Ende des mit `new` eingeleiteten Ausdrucks festgelegt.

Warnungen

- Die Deklaration des Klassennamens einer öffentlichen Klasse muss immer identisch sein zu dem Namen, unter dem der Source (Quellcode) dieser Datei gespeichert wird (ohne die Erweiterung `.java`). Aber auch für nichtöffentliche Klassen macht es Sinn, eine Datei, in der ausschließlich diese Klasse definiert ist, mit dem Klassennamen und der Erweiterung `.java` zu bezeichnen.
- Die Deklaration einer Klasse als abstrakt bedeutet nicht, dass vollständiger Code einen Fehler erzeugt. Eine abstrakte Klasse muss keinen unvollständigen Code enthalten – es ist nur wenig sinnvoll, solchen Code als abstrakt zu kennzeichnen.

- Die Modifier `final` und `abstract` dürfen nicht zusammen bei einer Klasse verwendet werden.
- Java verzichtet explizit auf Mehrfachvererbung.
- Meist sind Konstruktoren `public`. Sie dürfen nicht als `native`, `abstract`, `static`, `synchronized` oder `final` deklariert werden.
- Konstruktoren kann man nur überladen, aber nicht überschreiben.
- Es gibt in Java keine Destruktoren.

Beispiele

Eine einfache Klassendeklaration:

```
class Gaus
{
}
```

Eine als `public` deklarierte Klasse:

```
public class Teile
{
}
```

Eine als `public` deklarierte Klasse mit Kontruktormethode:

```
public class Teile
{
  Teile()
  {
   ...
   }
}
```

Eine Klassendeklaration mit Implementation einer Schnittstelle:

```
class MeinFenster implements ActionListener
{
}
```

Eine innere Klasse:

```
class AusKlasse
{
public class InKlasse
{
      ...
}
...
 }
```

Eine Klassendeklaration mit Angabe einer Superklasse:

```
public class MeinApplet extends Applet
{
}
```

Methoden

Methoden realisieren die Funktionalität von Objekten, ihre Implementierung erfolgt in Klassen.

Beschreibung

Im Rahmen der objektorientierten Programmierung werden Objekte als Zusammenschluss von zwei Bestandteilen gesehen, den sogenannten Objektdaten – das sind die Attribute beziehungsweise Eigenschaften – und den Objektmethoden. Attribute unterscheiden Objekte voneinander, während Objektmethoden die objektorientierte Ausdrucksweise für die Elemente der Anweisungenebene (Funktionen und Prozeduren) darstellen, d.h. Methoden sind die programmtechnische Umsetzung der Objektfunktionalitäten.

Anwendung

Methoden sind Algorithmen, mit denen Objekte miteinander kommunizieren und/oder ihre Objektdaten manipulieren können. Nur darüber können in Java Objekte Daten anderer Objekte lesen oder verändern. Es gibt in Java keine Objektmethoden ohne zugehörige Objekte. Auch die sogenannten Klassenmethoden liegen nicht außerhalb von Objekten, denn diese Methoden befinden sich aufgrund des Java-Meta-Klassenkonzeptes innerhalb einer Meta-Klasse. Die Deklaration einer Methode in Java sieht generell folgendermaßen aus:

```
[Zugriffsspezifizier] [modifier] returnwert nameMethode([Parameter]) [throws]
[ExceptionListe]
```

Dabei ist alles in eckigen Klammern optional.

Eine konkret verwendete Kombination aus Teilen der Definition wird Methodenunterschrift beziehungsweise Methodensignatur genannt. Über den Zugriffsspezifizierer `modifier` kann der Zugriff auf eine Methode für fremde Objekte/Klassen beschränkt werden. Voreingestellt ist immer der Default-Status freundlich, was bedeutet, dass diese Methoden frei innerhalb der Klasse, als auch des zugehörigen Paketes benutzt werden können. Der Zugriffsspezifizierer `public` hingegen legt fest, dass alle Objekte auf die Methode zugreifen können, egal in welchen Klassen oder Paketen sie definiert sind. Also eine Erweiterung der Zugrifferlaubnis. Der Zugriffsspezifier `protected` schützt Methoden gegenüber der Verwendung in fremden Paketen. Allerdings haben die Subklassen immer noch den vollen Zugriff auf ihre Superklassen. Dies ist also – entgegen der Suggestion durch den Namen – eine Erweiterung der freundlichen Voreinstellung und nur eine Einschränkung gegenüber dem öffentlichen Status. Die höchste auf eine Methode anwendbare Sicherheitsstufe kann durch die Benutzung von dem Zugriffsspezifiererer `private` eingerichtet werden. Eine private Methode ist nur für die anderen Methoden in derselben Klasse verfügbar. Sogar eine Subklasse dieser Klasse kann auf eine

private Methode nicht zugreifen. Sofern man allerdings die Modifier `private` und `protected` in Kombination verwendet, sind Methoden in der eigenen Klasse als auch für deren Subklassen (unabhängig von den Paketen, in denen sich Sub- und Superklasse befinden) verfügbar, aber nicht für den Rest des Pakets oder für Nichtnachfahrklassen außerhalb des Pakets. Bei dieser Kombination sollte aber beachtet werden, dass sie eigentlich nur im JDK 1.0 verwendet wurde (und schon da nicht perfekt funktioniert hatte) und in der Java-2-Plattform nicht mehr unterstützt wird[5].

Die Methodenmodifier ermöglichen es, bestimmte Eigenschaften einer Methode in Bezug auf die Sichtbarkeit und die Interaktion mit Subklassen festzulegen. Der Modifier `static` legt fest, dass es sich bei einer Methode um eine Klassenmethode handelt. Die Deklaration von Instanz- und Klassenmethoden unterscheidet sich ausschließlich durch das Plazieren dieses Modifiers am Beginn einer Methodendeklaration. Java unterscheidet aufgrund des objektorientierten Prinzips streng zwischen den Eigenschaften/Methoden einer spezifischen Instanz einer Klasse und der Klasse selbst. Eine Klassenmethode liegt außerhalb der konkreten Instanzen der Klasse, aber innerhalb der Klasse. Klassenmethoden werden über das Meta-Klassenkonzept in das streng objektorientierte Konzept von Java eingebunden. Methoden einer Klasse sind global, betreffen also immer die gesamte Klasse und sämtliche Instanzen der Klasse. Sie sind dort und in anderen Klassen verfügbar. Klassenmethoden können deshalb an beliebigen Stellen genutzt werden, unabhängig davon, ob eine konkrete Instanz der Klasse existiert oder nicht.

Der Modifier `abstract` legt fest, dass eine Methode zwar deklariert, aber nicht in der aktuellen Klasse implementiert wurde. Der nachfolgende Körper der Methode besteht ausschließlich aus einem einfachen Semikolon. Daher muss eine abstrakte Methode in der Subklasse der aktuellen Klasse überschrieben und implementiert werden, bevor mit der Methode irgendetwas anzufangen ist.

Der Modifier `final` verhindert, dass Subklassen einer Klasse diese Methode überschreiben können.

Der Modifier `native` deklariert Methoden, die nicht in Java geschrieben sind, aber dennoch innerhalb von Java verwendet werden sollen. Der Modifier `native` wird vor der Methode angegeben, und der Körper der Methode durch ein Semikolon ersetzt.

Über den Modifier `synchronized` lassen sich verschiedene Methoden im Rahmen des Multithreadingkonzeptes synchronisieren.

Der Rückgabewert einer Methode ist eine wichtige Information, ob die Methode korrekt gearbeitet hat, was sie genau ausgeführt hat oder sogar der eigentliche Zweck der Methode (etwa ein Objekt erzeugen). Rückgabewerte von Java-

5. Eine Verwendung ist unter der Java 2-Plattform nur noch über recht gefährliche Tricks möglich, auf die wir an anderer Stelle noch genauer eingehen wollen.

Methoden können daher von jedem erlaubten Datentyp sein, also sowohl einfache Objekte als auch Klassen (beispielsweise Zeichenketten). Eine Methode muss immer einen Wert zurückgeben (und zwar genau den Datentyp, welcher in der Deklaration angegeben wurde), es sei denn, sie ist mit dem Schlüsselwort `void` deklariert worden oder sie ist ein Konstruktor einer Klasse (wobei auch ein Konstruktor natürlich etwas zurückgibt – eine Instanz der Klasse, aber das wird nicht mehr in der Deklaration dokumentiert und nicht im Kontruktorkörper ausgewiesen). Rückgabewerte werden im Körper der Methode mit der Anweisung `return()`zurückgegeben. Innerhalb der Klammer steht der gewünschte Rückgabewert.

Bezüglich des Methodennamens gelten die gleichen Regeln wie bei allen Token.

Die optionale Parameterliste einer Methode ist eine Reihe von Informationen, welche in Klammern eingeschlossen an die Methode weitergegeben werden. Sie hat immer folgende Struktur:

```
Datentyp Variablenname, Datentyp Variablenname, Datentyp Variablenname, ...
```

Die Anzahl der Parameter ist beliebig und kann Null sein. In letzterem Fall sind die Klammern leer.

Der Aufruf einer Methode an anderer Stelle im Quellcode folgt der in Java üblichen DOT-Notation, die wir etwas weiter oben erläutert haben (siehe Seite 48).

Warnungen

- Weder statische Methoden noch Klassenkonstruktoren dürfen als abstrakt deklariert werden.
- Abstrakte Methoden können nie als `final` deklariert werden, weil dadurch verhindert wird, dass diese Methoden überschrieben werden können. Sie wären nutzlos.
- Sofern eine Klasse nicht alle abstrakten Methoden implementiert, muss auch sie als abstrakt deklariert werden.
- Java erlaubt ein Überladen von Methoden, jedoch kein Überladen von Operatoren.

Beispiele

Eine Initialisierungsmethode eines Applets:

```
public void init()
{
...//tue was
}
```

Eine Startmethode eines Applets:

```
public void start()
{
...//tue was

}
```

Eine öffentliche Klassenmethode – die Standardeinstiegsmethode jeder Java-Anwendung:

```
public static void main (String args[])
{
...//tue was
}
```

Objekte

Aus Klassen erzeugte reale Instanzen.

Beschreibung

Java verfolgt streng das Prinzip der objektorientierten Programmierung. In der objektorientierten Programmierung ist die Trennung von Datenebene und Anweisungenebene aufgehoben. Zusammengehörende Anweisungen und Daten bilden zusammengehörende, abgeschlossene und eigenständige Einheiten, die Objekte oder Instanzen, welche aus Klassen erzeugt werden.

Anwendung

Objekte werden in der Regel (aber nicht ausschließlich, Objekte können auch implizit entstehen – etwa beim Zusammenfügen von zwei Strings mit dem +-Operator oder als Rückgabewerte von Methoden) in Java über das Schlüsselwort `new` und dem nachfolgenden Aufruf einer Konstruktormethode aus einer Klasse erzeugt. Objekte bestehen im Allgemeinen aus zwei Bestandteilen, den Objektdaten – die Attribute beziehungsweise Eigenschaften – und den Objektmethoden. Objektdaten sind die Dinge, durch welche sich ein Objekt von einem anderen unterscheidet. Objektmethoden sind irgendwelche Algorithmen, mittels denen die – im Prinzip abgeschlossenen und eigenständigen – Objekte miteinander kommunizieren und/oder ihre Objektdaten manipulieren können. In Java können Objekte ausschließlich über Methoden kommunizieren. Dies schließt als wichtige Konsequenz beispielsweise eine Existenz von globalen Variablen aus (obwohl Klassenvariablen in gewisser Weise als Ersatz für einige Konstellationen fungieren können). Prinzipiell gibt es in Java weder Attribute noch Methoden, welche außerhalb von Objekten existieren. Damit Objekte aktiv werden können, tauschen sie sogenannte Botschaften aus, welche ausschließlich die Kommunikation von Objekten regeln. Andere Kommunikationswege gibt es nicht. Das sendende Objekt schickt dem Zielobjekt eine Aufforderung, eine bestimmte Aktion auszuführen. Das Zielobjekt reagiert entsprechend. Die genaue formale Schreibweise folgt in Java dem Schema »Empfänger Methodenname Argument«. Punkt und Klammer trennen dabei meist die drei Bestandteile der Botschaft (die sogenannte Punkt-Notation oder DOT-Notation) in der Form `Empfänger.Methodenname(Argument)`.

Nach außen ist ein Objekt nur durch seine Schnittstellen zu den Methoden definiert, es ist gekapselt, versteckt seine innere Struktur vollständig vor anderen Objekten (Information Hiding). Der ganz entscheidende Vorteil von diesem Verfahren ist, dass ein Objekt sich im Inneren, das heisst bezüglich seiner Objektdaten und seiner inneren Attribute, vollständig verändern kann. Solange es sich nur nach außen unverändert zeigt, wird das veränderte Objekt problemlos in ein System integriert, wo es in seiner bisherigen inneren Form funktioniert hatte.

Beispiele

Erzeugung eines Objektes mittels des Schlüsselwortes `new`:

```
URL  neueURL = new URL(UrlName1);
```

Implizite Erzeugung eines Objektes

```
String name;
Name = "Hans " + "Dampf";
```

Pakete

Wenn unter Java Klassen und Schnittstellen zu einer gemeinsamen Struktur zusammengefasst werden sollen, ordnet man sie einem gemeinsamen Paket zu.

Beschreibung

Pakete (Packages) sind in Java Gruppierungen von Klassen und Schnittstellen, eine spezielle Art des Designs und der Organisation von Java im Großen. Sie sind eine Art Java-Analogon zu Bibliotheken vieler anderer Computersprachen. Die gesamte Java-Laufzeitbibliothek, das Java-API, wird in Form von Paketen zur Verfügung gestellt. Ein Java-Paket enthält normalerweise logisch zusammenhängende Klassen und Schnittstellen.

Anwendung

Pakete als Sprachgruppierung von Java im Großen werden in der Regel so organisiert wie die Vererbungshierarchie. Die Java-Klassenbibliothek ist selbst so organisiert. Die oberste Ebene dort heisst `java`. Danach folgen Ebenen wie etwa `io`, `net`, `util` oder `awt`, denen unter Umständen weitere Ebenen folgen (bei `awt` etwa das Paket `image`).

Bezüglich der Namenskonventionen für Pakete gelten die üblichen Regeln für Token. Die erste Ebene der Hierarchie spezifiziert oft den eindeutigen Namen der Organisation, welche das Paket entwickelt hat. Es gibt aber auch Ausnahmen, wie das Standardpaket `java`.

Für Anwender gibt es zwei Berührungspunkte zur Paketstruktur von Java. Entweder er verwendet ein oder mehrere Paket(e) oder er erstellt ein Paket.

Die Erstellung eines Paketes bedeutet, dass ein Entwickler Klassen und Schnittstellen zu größeren Strukturen – Paketen – zusammenfassen möchte. Eine Java-Datei wird zu einem Paket beziehungsweise einem bestehenden Paket zugeordnet, wenn am Anfang der Datei als erste gültige Anweisung das Schlüsselwort `package`, gefolgt von dem Namen des Paketes, und einem abschließenden Semikolon steht. Anschließend werden einfach Klassen beziehungsweise Schnittstellen notiert. Wenn sich innerhalb einer Quelldatei mehrere Klassendefinitionen befinden, dann werden alle Klassen dem durch das Schlüsselwort `package` angegebenen Paket zugeordnet. Sofern die `package`-Anweisung fehlt, wird die Klasse einem voreingestellten Paket ohne Namen (einem sogenannten anonymen Paket) zugeordnet. Die Klassen dieses Paketes können dann direkt von allen Klassen importiert werden, welche im gleichen Verzeichnis stehen (und nur von diesen). Hierbei wird dann die `import`-Anweisung ohne weitere Qualifizierung angegeben.

Auch die allgemeine Verwendung von Paketen innerhalb von Klassen basiert auf dieser `import`-Anweisung. Zwar kann jede (zugängliche) Komponente einer anderen Klasse über eine vollständig qualifizierte Punktnotation angesprochen werden. Dies erweist sich aber weder in Bezug auf Lesbarkeit des Quelltextes und vor allem Schreibaufwand als wenig praktikabel. Die `import`-Anweisung bedeutet eine einfachere und schnellere Technik zur Referenzierung. Eine `import`-Zeile, die vor der Definition irgendeiner Klasse in der Datei stehen muss, importiert das notierte Element. Es kann durchaus mehrfache `import`-Anweisungen innerhalb einer Klassendefinition geben. Wenn in einem Paket selbst ein anderes Element importiert werden soll, muss jede notwendige `import`-Anweisung nach der `package`-Anweisung stehen.

Die `import`-Anweisung dient in Java nur dazu, Elemente über einen verkürzten Namen innerhalb der aktuellen Bezugsklasse zugänglich zu machen und damit den Code zu vereinfachen. Sie hat nicht den Sinn, die Elemente einzulesen.

Es lässt sich mit einer `import`-Anweisung sowohl eine einzelne Klasse als auch ein ganzes Paket importieren. Eine einzelne Klasse wird über einen vollständig qualifizierten Namen (im Sinne der Paketstruktur) importiert, bei Import von einem vollständigen Paket wird der Klassenname durch eine Wildcard ersetzt. Der Platzhalter ist wie oft üblich ein Stern (*). Der `import`-Befehl kennt noch eine dritte Variante. Dabei wird nur ein Teil des vollständig qualifizierten Namens angegeben und ohne Platzhalter mit einem Semikolon abgeschlossen. Dann müssen aber bei einer Verwendung das letzte Element aus dem importierten Packagenamen und die noch fehlenden Angaben des vollständig qualifizierten Namens per Punktnotation notiert werden.

Warnungen

Das Sternchen importiert keine untergeordneten Pakete. Um also alle Klassen einer komplexen Pakethierarchie zu importieren, muss explizit für jede Hierarchieebene eine `import`-Anweisung stehen.

Beispiele

Die Erstellung eines eigenen Paketes;

```
package MeinErstesPaket;
public class MeineErsteKlasse
{
...
}
...
```

Verschiedene Versionen des Importierens:

```
import EinPackage.MeineKlasse;
import EinPackage.*;
import EinPackage;
```

Schnittstellen

Eine Schnittstelle ist unter Java eine Sammlung von Konstanten beziehungsweise Methoden-Namen ohne konkrete Definition.

Beschreibung

Schnittstellen sind in Java eine Ersatztechnologie für Mehrfachvererbung. Mittels Schnittstellen können Methoden bezüglich ihrer Methodenunterschrift (Sichtbarkeit, Zugriffsregeln, Rückgabewert, Name, Parameter) festgelegt und ein Anwender der Schnittstelle gezwungen werden, sämtliche dort deklarierten Methoden in der implementierenden Klasse zu überschreiben, also die entsprechenden Schritte für eine vollständige Funktionalität einer Aktion explizit zu handhaben. Daneben lassen sich in Schnittstellen global verwendbare Konstanten deklarieren.

Anwendung

Klassen beziehungsweise Objekte können in Java eine beliebige Anzahl an Schnittstellen implementieren. Dies ist besonders von Bedeutung, weil eine einzelne Java-Klasse nur genau eine Superklasse haben kann (und auch genau eine haben muss!). Java-Schnittstellen sind wie die IDL(Interface Description Language)-Schnittstellen – ein Schnittstellenstandard zum Informationsaustauch verschiedener Programmiersprachen – aufgebaut. Schnittstellen ermöglichen, ähnlich wie abstrakte Klassen, das Erstellen von Pseudoklassen, die ganz aus abstrakten Methoden zusammengesetzt sind. Durch die Implementierung von Schnittstellen muss jede nicht-abstrakte Klasse alle in der Schnittstelle deklarierten Methoden überschreiben. Dies bedeutet, dass eine vollständige Schnittstelle nicht verwendet werden kann, bis alle darin enthaltenen Methoden in der implementierenden Klasse überschrieben sind!

Die Implementierung von Schnittstellen in Klassen erfolgt über das in der Klassendeklaration angegebene Schlüsselwort `implements`. Wenn mehr als eine Schnittstelle implementiert werden soll, müssen die einzelnen Schnittstellennamen in der Schnittstellenliste der Schnittstellendeklaration durch Kommata getrennt werden.

Wenn in einer Klasse eine Schnittstelle implementiert werden soll, muss die entsprechende Klasse bestimmte Voraussetzungen erfüllen. Alle in einer Schnittstelle deklarierten Methoden sind als Grundeinstellung mit dem Zugriffslevel `public` ausgestattet. Eine solche Methode kann in der implementierenden Klasse nicht so überschrieben werden, dass der Zugriff auf sie beschränkt wird. Deshalb müssen alle in einer Schnittstelle deklarierten und in einer Klasse überschriebenen Methoden ebenfalls mit dem Zugriffsmodifier `public` versehen werden. Von den übrigen Modifizierern, die auf Methoden angewendet werden können, dürfen nur `native` und `abstract` auf solche Methoden angewendet werden, die ursprünglich in einer Schnittstelle deklariert wurden.

Schnittstellenmethoden können eine Parameterliste mit Angaben definieren, die an die Methode weitergegeben werden müssen. Wenn in der Klasse eine neue Methode mit dem gleichen Namen, jedoch einer anderen Parameterliste deklariert wird, wird wie allgemein üblich die in der Schnittstelle deklarierte Methode überladen und nicht überschrieben. Dies ist zwar nicht falsch, aber dann muss noch zusätzlich die Methode überschrieben werden, da jede Methode in einer Schnittstelle explizit überschrieben werden muss (außer sie wird als abstrakt deklariert).

Eine konkrete Schnittstellendeklaration hat viel Ähnlichkeit zu einer Klassendeklaration. Sie besteht aus fünf Teilen, welche teilweise optional sind. Begonnen wird eine Schnittstellendeklaration mit optionalen Schnittstellenmodifier. Dies sind die Schlüsselworte `abstract` oder `public`. Dabei sollte beachtet werden, dass alle Schnittstellen abstrakt sind und dieser Modifier deshalb nicht unbedingt notwendig ist. Der Modifier kann jedoch trotzdem zur Eindeutigkeit gesetzt werden. Das nachfolgende Schlüsselwort `interface` ist zwingend und gibt an, dass es sich um eine Schnittstelle handelt. Ebenso ist der Bezeichner selbst zwingend. Die Namensregeln folgen den üblichen Regeln für Token und die üblichen Konventionen für Schnittstellennamen sind denen von Klassen gleich (erster Buchstabe jedes Schnittstellennamens beziehungsweise Teilnamens groß). Die nachfolgende Schnittstellenerweiterung ist hingegen wieder optional. Sie wird durch das Schlüsselwort `extends`, gefolgt von einer durch Kommata getrennten Liste von Schnittstellenbezeichnern, deklariert. Der Schnittstellenkörper ist wieder zwingend. Da eine Schnittstelle nur abstrakte Methoden und finale Variablen enthalten darf, können im Körper einer Schnittstelle keine konkreten Implementierungen spezifiziert werden. Methoden in Schnittstellen können also keinen Körper haben, eine Schnittstellenmethode besteht nur aus einer Deklaration. Die Deklaration einer Methode in Schnittstellen endet also direkt mit einem Semikolon.

Schnittstellen können als Voreinstellung von allen Klassen im selben Paket implementiert werden (freundliche Defaulteinstellung). Damit verhalten sie sich wie freundliche Klassen und Methoden. Indem eine Schnittstelle explizit als `public` deklariert wird, kann diese Schnittstelle außerhalb eines gegebenen Pakets implementiert werden.

Java-Schnittstellen können auch andere Schnittstellen erweitern (die Angabe `extends` in der Deklaration einer Schnittstelle). Die implementierende Subschnittstelle erbt dabei auf die gleiche Art und Weise wie Klassen die Eigenschaften von der(n) Superschnittstelle(n). Vererbt werden alle Methoden und statischen Konstanten der Superschnittstelle. Eine Klasse, welche die Subschnittstelle implementiert, muss sowohl die in der Subschnittstelle deklarierten Methoden, als auch alle Methoden der Superschnittstellen überschreiben.

Variablen in Schnittstellen sind immer Konstanten. Sie sind, unabhängig vom Modifier, der bei der Deklaration des Feldes benutzt wurde, immer `public`, `final` und `static`. Diese Angaben müssen in der Deklaration nicht explizit angegeben

werden, es ist der besseren Lesbarkeit halber jedoch sinnvoll. Weil alle Variablen in Schnittstellen final sind, müssen sie in der Schnittstelle unbedingt initialisiert werden. Der Compiler meldet bei einer fehlenden Wertzuweisung einen Fehler.

Ausnahmen in Schnittstellen spielen eine relativ geringe Rolle. Zwar kann bei der Deklaration einer Methode in einer Schnittstelle eine ausgeworfene Ausnahme angegeben werden. Da es sich jedoch nur um abstrakte Methoden handelt, ist diese Angabe alleine für das Überschreiben von Schnittstellenmethoden von Bedeutung. Dabei darf die neue Ausnahmenliste in der späteren Implementierung nur Ausnahmen enthalten, die auch in der ursprünglichen Ausnahmenliste oder in deren Subklassen vorhanden sind. Es gilt sogar, dass die neue Ausnahmenliste nicht unbedingt Ausnahmen enthalten muss, egal wie viele in der ursprünglichen Liste vorhanden sind. Der Grund ist, dass die alte Liste der neuen Methode durch Vererbung zugewiesen wird. Die überschriebene Methode kann jede in der ursprünglichen Ausnahmenliste enthaltenen oder aus dieser Liste abgeleiteten Ausnahmen, unabhängig von der eigenen Ausnahmenliste, auswerfen. Im Allgemeinen jedoch bestimmt die Ausnahmenliste der in der Schnittstelle deklarierten Methode, und nicht die der redeklarierten Methode, welche Ausnahmen aufgeworfen werden können und welche nicht.

Warnungen

- Analog `public`-Klassen müssen als `public` deklarierte Schnittstellen zwingend in einer Datei namens `<NamederSchnittstelle>.java` definiert werden.
- Andere Zugriffsmodifier als das Schlüsselwort `public` sind bei der Deklaration einer Methode in einer Schnittstelle nicht erlaubt.
- Da alle Variablen in Schnittstellen final sind, müssen sie in der Schnittstelle unbedingt initialisiert werden. Der Compiler meldet bei einer fehlenden Wertzuweisung einen Fehler.
- Da Körper von Schnittstellenmethoden leer sind, muss als wesentliche Aufgabe bei der Implementierung einer Schnittstelle für jede ursprünglich in der Schnittstelle deklarierte Methode in einer Klasse ein Körper für die ursprünglich in der Schnittstelle deklarierten Methoden erstellt werden (außer, die Methoden sollen nativ oder die neue Klasse abstrakt sein). Bezüglich des tatsächlich realisierten Methodenkörpers in der implementierenden Klasse bestehen wenig Restriktionen. Die Erstellung eines Methodenkörpers, welcher nur aus geöffneten und geschlossenen geschweiften Klammern besteht, reicht aus, um die Bedingung für die Implementation einer Schnittstellenmethode zu erfüllen, deren Rückgabetyp `void` ist. Bei Methoden mit anderem Rückgabetyp langt im Methodenkörper eine entsprechende `return()`-Anweisung. Sinnvoll ist dies jedoch kaum. Zwar wird damit der Compiler insofern ausgetrickst, als dass dieser annimmt, die Bedingungen für die Implementierung einer

Schnittstelle sind erfüllt. Allerdings werden Methoden in Schnittstellen nicht ohne zwingenden Grund auf eine bestimmte Art deklariert und eine fehlende Funktionalität kann das Resultat negativ verändern.

- Schnittstellen können nicht per `implements` weitere Schnittstellen implementieren. Dafür wird in der Schnittstellendeklaration das Schlüsselwort `extends` verwendet, das auch beim Ableiten von einer Klasse verwendet wird. Dies ist besonders gefährlich, weil in einer Klassendeklaration die Bedeutungen abweichen.

Beispiele

Deklaration einer Schnittstelle, wo nur zwei Konstanten deklariert werden, die dann in einer Klasse ausgegeben werden:

```
interface MeineSchnittstelle
{
 int zaehler = 42;
 String str = "Test in der Schnittstelle.";
...
}
public class InterfaceTest implements MeineSchnittstelle
{
...
public static void main (String args[])
 {
...
  System.out.println(zaehler); //Ausgabe
  System.out.println(str); //Ausgabe
...
 }
}
```

Ein Beispiel mit zwei Schnittstellen, welche beide in einer Klasse implementiert werden. Die zweite Schnittstelle enthält eine Methodendeklaration, welche zwingend in der implementierenden Klasse überschrieben werden muss:

```
interface MeineSchnittstelle
{
 int zaehler = 42;
 String str = "Test in der Schnittstelle.";
...
}
interface MeineZweiteSchnittstelle
{
...
 public void meineMethode();
}
```

```
class MethodenAufruf implements MeineSchnittstelle, MeineZweiteSchnittstelle
{
...
 public void meineMethode()
 {
...
  System.out.println(zaehler); //Ausgabe
  System.out.println(str); //Ausgabe
 }
}
```

Token

Ein Token bezeichnet ein Zeichen oder Merkmal, verwendet im Rahmen einer Sprache, als Sinnzusammenhang.

Beschreibung

Wenn ein Compiler eine Datei übersetzt, muss er zunächst herausfinden, welche einzelnen Zeichen und Symbole oder zusammennotierten Zeichen- und Symbolgruppen im Code welche Bedeutung haben. Ein Token ist am besten als Sinnzusammenhang zu verstehen. Ein Token ist dabei gelegentlich ein bestimmtes einzelnes Zeichen/Symbol oder meistens ein aus mehreren Zeichen zusammengesetzter Begriff. Das interpretierende System versteht ein gültiges Zeichen oder die Sammlung von Zeichen. Erkennt das System das spezifische Zeichen oder die Ansammlung von Zeichen, wird daraus ein Sinnzusammenhang, dem eine bestimmte Aktion durch das System folgt. Sonst bleibt das spezifische Zeichen oder die Ansammlung von Zeichen einfach nur die Summe der Buchstaben oder Zeichen.

Anwendung

Wenn der Quellcode kompiliert wird, zerlegt der Java-Compiler ihn in einzelne kleine Bestandteile (Token). Die Sprachelemente werden dabei auf ihre Richtigkeit geprüft und – falls alles in Ordnung ist – folgt bei Erkennen eine sinnvolle Reaktion durch das System. Es gibt in Java folgende Arten von Token (beachten Sie, dass wir Leerräume und Kommentare in die Tabelle aufgenommen haben, obwohl sie technisch gesehen keine Token sind):

Bezeichner	Ein Bezeichner oder Identifier ist in Java ein gültiger Name (eine Zeichenkette) für Klassen, Objekte, Variablen, Konstanten, Bezeichnungsfelder, Methoden etc. Zusammengesetzt werden Bezeichner aus alphanumerischen Unicode-Zeichen. Sie dürfen im Prinzip eine unbeschränkte Länge haben (bis auf technische Einschränkungen durch das Computersystem). Dabei müssen folgende Regeln eingehalten werden: • An der ersten Stelle eines Bezeichners darf keine Zahl stehen. Das erste Zeichen eines Bezeichners muss ein Buchstabe, der Unterstrich (_) oder das Dollarzeichen ($) sein. Alle folgenden Zeichen müssen entweder Buchstaben oder Zahlen sein. Es müssen jedoch nicht unbedingt Zeichen des lateinischen Alphabetes oder Zahlen sein. Zeichen jeden Alphabetes, das von Unicode unterstützt wird, sind erlaubt. • Bezeichner dürfen nicht mit Java-Schlüsselworten und sollten nicht mit Namen von Java-Paketen identisch sein.

Neben den zwingenden Namensregeln gibt es in Java einige Namenskonventionen, welche zwar nicht zwingend, aber bei Bezeichnern allgemein üblich sind:

- Möglichst aussagekräftige Bezeichner verwenden. Ausnahmen sind Schleifen, wo meistens nur ein Buchstabe als Zählvariable (normalerweise i oder j) verwendet wird.
- Konstanten sollten vollständig groß geschrieben werden.
- Die Identifier von Klassen sollten mit einem Großbuchstaben beginnen und anschließend klein geschrieben werden. Wenn sich ein Bezeichner aus mehreren Worten zusammensetzt, dürfen diese nicht getrennt werden. Die jeweiligen Anfangsbuchstaben werden jedoch innerhalb des Gesamtbezeichners jeweils groß geschrieben.
- Die Identifier von Variablen, Methoden und Elementen beginnen mit Kleinbuchstaben und werden auch anschließend klein geschrieben. Wenn sich ein Bezeichner aus mehreren Worten zusammensetzt, dürfen diese nicht getrennt werden. Die jeweiligen Anfangsbuchstaben von den folgenden Worten werden jedoch innerhalb des Gesamtbezeichners jeweils groß geschrieben.

Wenn es in einem Quelltext zwei identische Bezeichner gibt, stellt Java eine Technik zur Verfügung, mit der Namenskonflikte aufgelöst werden können – die sogenannten Namensräume. Dies sind Bereiche, in dem ein bestimmter Bezeichner benutzt werden kann. Namensräume sind in Java einer Hierarchie zugeordnet. Es gilt dabei die Regel, dass ein Bezeichner einen identischen Bezeichner in einem übergeordneten Namensraum überdeckt. Außerdem trennt Java die Namensräume von lokalem und nicht- lokalem Code. Die Hierarchie der Namensräume gliedert sich wie folgt:

- Links außen steht der Namensraum des Pakets, zu dem die Klasse gehört.
- Danach folgt der Namensraum der Klasse.
- Es folgen die Namensräume der einzelnen Methoden. Dabei überdecken die Bezeichner von Methodenparametern die Bezeichner von Elementen der Klasse. Sollten Elemente der Klasse überdeckt werden, können sie immer noch über das Schlüsselwort `this` angesprochen werden.

Innerhalb von Methoden gibt es unter Umständen noch weitere Namensräume in Form von geschachtelten Blöcken. Variablen, welche innerhalb eines solchen geschachtelten Blocks deklariert werden, sind außerhalb des Blocks unsichtbar.

Die Auflösung von Namensräumen bei Bezeichnern erfolgt immer von links nach rechts. Wenn der Compiler einen Bezeichner vorfindet, wird er zuerst im lokalen Namensraum suchen. Sofern er dort nicht fündig wird, sucht er im übergeordneten Namensraum. Das Verfahren setzt sich analog bis zum ersten Treffer (gegebenenfalls bis zur obersten Ebene) fort. Es gelten immer nur die Vereinbarungen des Namensraums, wo der Treffer erfolgt ist.

Schlüsselworte	Schlüsselworte sind alle Worte, die ein essenzieller Teil der Java-Sprachdefinition sind. Darauf gehen wir in dem nachfolgenden Kapitel noch intensiv ein.
Literale	Mit einem Literal können Variablen und Konstanten bestimmte Werte zugewiesen werden. Dies können sämtliche in der Java-Sprachdefinition erlaubten Arten von Werten sein (numerische Werte, boolesche Werte, Buchstaben, Zeichenketten). Die Literale werden nach ihrem Typ unterschieden.
	Ganzzahlliterale Ganzzahlliterale beziehungsweise Integerliterale sind Werte vom Datentyp `int` oder `long`. Sie können dezimal, aber auch hexadezimal sowie oktal beschrieben werden. Die Voreinstellung ist dezimal und der Datentyp `int`. Diese Defaulteinstellung gilt immer dann, wenn die Werte ohne weitere Angaben dargestellt werden. Hexadezimale Darstellungen beginnen immer mit der Sequenz `0x` oder `0X`. Oktale Darstellungen beginnen mit einer führenden Null. Negativen Ganzzahlen wird einfach ein Minuszeichen vorangestellt. Durch Anhängen von `l` oder `L` kann der Datentyp `long` gewählt werden. Sofern man für einen `int`-Datentyp einen Wert wählt, welcher den zulässigen Wertebereich überschreitet, muss dies sogar erfolgen.
	Gleitpunktliterale Gleitpunktliterale beziehungsweise Gleitzahlliterale sind Literale vom Datentyp `float` oder `double`. Der Dezimalpunkt trennt Vor- und Nachkommateil der Gleitzahl. Standardeinstellung ist `double`. Wenn ein Gleitzahlliteral als `float` interpretiert werden soll, muss ein `f` oder ein `F` angehängt werden. Negativen Gleitzahlen wird ein Minuszeichen vorangestellt. Mit dem nachgestellten `e` oder `E`, gefolgt von einem Exponenten (ein negativer Exponent ist erlaubt), können für Gleitzahlliterale Exponenten verwendet werden. Allerdings nur dann, wenn kein Dezimalpunkt vorhanden ist.
	Zeichenliterale Zeichenliterale werden durch ein einzelnes, zwischen hochgestellte und einfache Anführungszeichen stehendes Zeichen ausgedrückt. Das gilt für alle Zeichenwerte, egal ob es sich dabei um ein lateinisches Zeichen oder irgendwelche anderen Unicode-Zeichen handelt. Als einzelne Zeichen gelten alle druckbaren Zeichen mit Ausnahme des Bindestrichs (-) und des Backslash (\). Die Zeichen werden in Unicode-Format gespeichert. Zeichenliterale lassen sich aber auch in Escape-Format darstellen. Die Escape-Zeichenliterale beginnen immer mit dem Backslash-Zeichen. Diesem folgt eines der Zeichen (`b,t,n,f,r,",' \`) oder eine Serie von Oktalziffern (3-stellig) oder ein `u`, gefolgt von einer vierstelligen Serie von Hexadezimalziffern, die für ein nicht zeilenbeendendendes Unicode-Zeichen stehen. Die vier Stellen der hexadezimalen Unicode-Darstellung (`\u0000` bis `\uFFFF`) erlauben 65535 Kodierungen. Damit ist es insbesondere möglich, solche Zeichen innerhalb von Zeichenketten darzustellen, welche ohne diese Maskierung eine besondere Funktion haben:

Escape	Unicode	Oktal	Bedeutung
`\b`	`\u0008`	`\010`	Rückschritt (Backspace)
`\t`	`\u0009`	`\011`	Tab
`\n`	`\u000a`	`\012`	Neue Zeile
`\f`	`\u000c`	`\014`	Formularvorschub (Formfeed)
`\r`	`\u000d`	`\015`	Wagenrücklauf (Return)
`\"`	`\u0022`	`\042`	Doppeltes Anführungszeichen
`\'`	`\u0027`	`\047`	Einfaches Anführungszeichen
`\\`	`\u005c`	`\134`	Backslash

Zeichenkettenliterale

Zeichenkettenliterale sind aus mehreren Zeichenliteralen zusammengesetzte Ketten (Strings). Bei Zeichenkettenliteralen werden null oder mehr Zeichen in Anführungszeichen dargestellt. Die eingeschlossenen Zeichen können auch Steuerzeichen wie Tabulatoren, Zeilenvorschübe, nichtdruckbare Unicode-Zeichen oder druckbare Unicode-Spezialzeichen, Escape- oder Oktal-Sequenzen sein. Java erzeugt Zeichenketten als Instanz der Klasse `String`. Damit stehen alle Methoden dieser Klasse zur Manipulation einer Zeichenkette zur Verfügung. Etwa zum Vergleichen oder Durchsuchen von Zeichenketten. Auch die Addition von Zeichenketten ist definiert. Mit dem Verknüpfungsoperator (+) kann eine neue Zeichenkette aus mehreren kleinen Zeichenketten zusammengesetzt werden. Wenn ein Wert, der keine Zeichenkette ist, mit einer Zeichenkette verbunden wird, so wird er vor dem Verbinden automatisch zu einer Zeichenkette konvertiert.

Boolesche Literale

Boolesche Literale kommen in Java nur als `true` und `false` vor. Es gibt keinen Nullwert und kein numerisches Äquivalent.

Null-Literal

Die `null`-Referenz wird in Java über das Literal `null` dargestellt.

Trennzeichen

Ein Trennzeichen ist eines jener Java-Symbole, die dazu benutzt werden, Trennungen und Zusammenfassungen von Code anzuzeigen. Es handelt sich um die folgenden Zeichen:

- `(` Öffnen einer Parameterliste für eine Methode, als auch Festlegung eines Vorrangs für Operationen in einem Ausdruck.
- `)` Schließen einer geöffneten Parameterliste für eine Methode, als auch zur Beendigung eines mit `)` festgelegten Vorrangs für Operationen in einem Ausdruck.
- `{` Beginn eines Blockes mit Anweisungen oder einer Initialisierungsliste.
- `}` Ende eines geöffneten Blockes mit Anweisungen oder einer Initialisierungsliste.
- `[` Das Token steht vor einem Ausdruck, der als Index für ein Feld dient.

	`]` Das Token folgt einem Ausdruck, der als Index für ein Feld dient, und beschließt den Index. `;` Beenden einer Ausdrucksanweisung, als auch zum Trennen der Teile bei einer for-Anweisung. `,` Multifunktionaler Begrenzer. `.` Dezimalpunkt oder Trennzeichen von Paket, Klassen, Methoden- und Variablennamen.
Operatoren	Operatoren sind Zeichen oder Zeichenkombinationen, die eine auszuführende Operation mit einer oder mehreren Variablen oder Konstanten angibt. In Java werden die Operatoren in mehreren Kategorien eingeteilt. Es gibt • fünf arithmetische Operatoren • sechs Zuweisungsoperatoren • einen Dekrementoperator • einen Inkrementoperator • vier bitweise arithmetische Operatoren • drei bitweise Verschiebungsoperatoren • sechs bitweise Zuweisungsoperatoren • sechs Vergleichsoperatoren • drei logische Vergleichsoperatoren und • zwei Operatoren, welche innerhalb `if-then-else`-Konstrukten als Ersatz für eine ausführliche Schreibweise fungieren, wenn sie zusammen benutzt werden **Arithmetische Operatoren** Arithmetische Operatoren benutzen nur zwei Operanden. Dies sind entweder ganzzahlige Werte oder Fließkommazahlen. Es gibt folgende arithmetischen Java-Operatoren: `+` Additionsoperator `-` Subtraktionsoperator `*` Multiplikationsoperator `/` Divisionsoperator `%` Modulooperator Rückgabe einer arithmetischen Operation ist ein Wert, dessen Datentyp sich aufgrund der Datentypen der Operanden wie folgt ergibt: • Zwei ganzzahlige Datentypen (`byte, short, int` oder `long`) als Operanden ergeben immer einen ganzzahligen Datentyp als Ergebnis. Dabei kann der Datentyp des Ergebnisses immer nur ein Datentyp `int` oder `long` sein. Datentyp `long` entsteht dann und nur dann, wenn einer der beiden Operanden bereits vom Datentyp `long` war, oder das Ergebnis von der Größe her nur als `long` dargestellt werden kann. • Zwei Fließkommatypen als Operanden ergeben immer einen Fließkommatypen als Ergebnis. Die Anzahl der Stellen des Ergebnisses ist immer das Maximum der Stellenanzahl der beiden Operanden.

- Wenn die Operanden ein ganzzahliger Typ und ein Fließkommatyp sind, ist das Ergebnis immer ein Fließkommatyp.

Einstellige arithmetische Operatoren

Einstellige arithmetische Operatoren verwenden nur einen Operanden. Dieser wird dem Operator nachgestellt. Es gibt in Java zwei:

- Einstellige arithmetische Negierung

\+ Die Umkehr der arithmetischen Negierung

Die einstellige Negierung ergibt die arithmetische Vorzeichenumdrehung ihres nachfolgenden numerischen Operanden. Der einstellige Operator + ist nur aus Symmetriegründen vorhanden.

Zuweisungsoperatoren

Zuweisungsoperatoren weisen das Ergebnis einer auf der rechten Seite stehenden Operation (damit ist auch eine Konstante gemeint) der linken Seite des Ausdrucks zu. In Java gibt es neben dem direkten Zuweisungsoperator (=) noch die arithmetischen Zuweisungsoperatoren. Diese sind Abkürzungen für arithmetische Operationen. Wie auch die arithmetischen Operatoren, können sie sowohl mit ganzen Zahlen als auch mit Fließkommazahlen verwendet werden. Es gibt folgende Zuweisungsoperatoren:

`+=` Additions- und Zuweisungsoperator

`-=` Subtraktions- und Zuweisungsoperator

`*=` Multiplikations- und Zuweisungsoperator

`/=` Divisions- und Zuweisungsoperator

`%=` Modulo- und Zuweisungsoperator

`=` Direkter Zuweisungsoperator

Inkrement-/Dekrementoperatoren

Inkrement-/Dekrementoperatoren werden zum Auf- und Abwerten eines einzelnen Wertes verwendet. Inkrement- und Dekrement-operatoren sind einstellige Operatoren und werden nur in Verbindung mit einem ganzzahligen oder einem Fließkommaoperanden benutzt.

Der Inkrementoperator (++) erhöht den Wert des Operanden um 1. Die Reihenfolge von Operand und Operator ist relevant. Wenn der Operator vor dem Operanden steht, erfolgt die Erhöhung des Wertes, bevor der Wert dem Operanden zugewiesen wird. Wenn er hinter dem Operanden steht, erfolgt die Erhöhung, nachdem der Wert bereits zugewiesen wurde.

Der Dekrementoperator (`--`) verringert den Wert des Operanden um 1. Die Reihenfolge von Operand und Operator ist auch hier relevant. Wenn der Operator vor dem Operanden steht, erfolgt die Verringerung des Wertes, bevor der Wert dem Operanden zugewiesen wird. Wenn er hinter dem Operanden steht, erfolgt die Verringerung, nachdem der Wert bereits zugewiesen wurde.

Bitweise arithmetische Operatoren

Bitweise arithmetische Operatoren verändern Bitwerte. In Java ist bitweise Arithmetik nur für die vier Integer-Typen und für Zeichentypen (`char`) definiert, nicht aber für boolesche Typen und Fließkommatypen. Der bitweise Komplementoperator (~) unterscheidet sich massiv von den anderen drei bitweisen Operatoren, denn Komplementieren ist eine einstellige bitweise Operation mit nachgestelltem Operanden (im Gegensatz zu den anderen drei Operationen, welche zweistellig sind). Wenn ein Byte komplementiert wird, werden alle seine Bits invertiert. Die Bitoperatoren von Java sind folgende:

`&` Bitweiser AND-Operator

`|` Bitweiser OR-Operator

`^` Bitweiser XOR-Operator

`~` Bitweiser Komplement-Operator

Bitweise Verschiebungsoperatoren

Bitweise Verschiebungsoperatoren verschieben die Bits in einer ganzen Zahl. Die Bits des ersten Operanden werden um die Anzahl an Positionen verschoben, die im zweiten Operanden angegeben wird. Im Fall der Verschiebung nach links ist es immer eine Null, mit der die rechte Seite aufgefüllt wird. Dieser Vorgang entspricht dem Multiplizieren mit 2 hoch der Zahl, die durch den zweiten Operanden definiert wird. Der normale Verschiebungsoperator nach rechts vervielfacht das Vorzeichenbit. Dieser Vorgang entspricht der Division durch 2 hoch der Zahl, die durch den zweiten Operanden definiert wird. Die Verschiebung nach rechts mit Füllnullen vervielfacht eine Null von der linken Seite. Java kennt folgende bitweisen Verschiebungsoperatoren:

`<<` Bitweise Verschiebung nach links

`>>` Bitweise Verschiebung nach rechts

`>>>` Bitweise Verschiebung nach rechts mit Füllnullen

Bitweise Zuweisungsoperatoren

Bitweise Zuweisungsoperatoren verwenden einen Wert, führen eine entsprechende bitweise Operation mit dem zweiten Operanden durch und legen das Ergebnis als Inhalt des ersten Operanden ab. Java kennt folgende bitweisen Zuweisungsoperatoren:

`&=` Bitweise AND-Zuweisung

`|=` Bitweise OR-Zuweisung

`^=` Bitweise XOR-Zuweisung

`<<=` Bitweise Verschiebungszuweisung nach links

`>>=` Bitweise Verschiebungszuweisung nach rechts

`>>>=` Bitweise Verschiebungszuweisung nach rechts mit Füllnullen

Vergleichsoperatoren

Vergleichsoperatoren verwenden zwei Operanden und vergleichen diese. Es werden entweder zwei Ganzzahlen oder zwei Fließkommazahlen verglichen.

Bei einem Vergleich von zwei Operanden vom Typ `char` werden diese wie ganze 16-Bit-Zahlen entsprechend ihrer Unicode-Kodierung behandelt. Als Rückgabewert der Operation entsteht ein boolescher Wert. Java kennt folgende Vergleichsoperatoren:

`==`	Gleichheitsoperator
`!=`	Ungleichheitsoperator
`<`	Kleiner-als-Operator
`>`	Größer-als-Operator
`<=`	Kleiner-als-oder-gleich-Operator
`>=`	Größer-als-oder-gleich-Operator

Logische Vergleichsoperatoren

Die logischen Vergleichsoperatoren werden nur auf boolesche Operanden angewandt und erzeugen auch nur boolesche Ergebnisse. Java kennt folgende logischen Vergleichsoperatoren:

`&&`	Logischer AND-Operator
`||`	Logischer OR-Operator
`!`	Logischer NOT-Operator

Der if-then-else-Operator

Unter dem if-then-else-Operator versteht man die Kombination der beiden Zeichen Fragezeichen und Doppelpunkt (`?:`). Der Doppeloperator arbeitet mit drei Operanden. Die Operation benötigt einen booleschen Ausdruck vor dem Fragezeichen. Wenn er `true` ist, wird der Wert vor dem Doppelpunkt zurückgegeben, ansonsten der Wert hinter dem Doppelpunkt.

Kommentare

Ein Kommentar (rein technisch betrachtet kein Token, aber vom logischen Gesichtspunkt dazupassend) dient zur übersichtlichen Gestaltung des Quellcodes. Der Compiler ignoriert Kommentare. Java kennt drei Arten von Kommentaren:

Zwei Schrägstriche, die den Rest einer Zeile als Kommentar kennzeichnen:

```
// Kommentar bis zum nächsten Zeilenende
```

Ein Bereich, der mit einem Schrägstrich und einem Stern begonnen und mit einem Stern und einem Schrägstrich beendet wird. Sämtlicher eingeschlossener Text wird als Kommentar gekennzeichnet:

```
/* Eingebetteter Kommentar */
```

Ein Bereich, der mit einem Schrägstrich und zwei Sternen begonnen und mit einem Stern und einem Schrägstrich beendet wird. Sämtlicher eingeschlossener Text wird als Kommentar gekennzeichnet:

```
/** javadoc-Kommentar */
```

Eine Besonderheit ist der `javadoc`-Kommentar, denn er wird vom Java-Dokumentations-Tool `javadoc` bei der Erstellung einer Dokumentation ausgewertet.

Leerräume	Ein Leerraum (Leerzeichen, Tabulator und Zeilenvorschub) ist – wie der Kommentar – rein technisch gesehen kein Token (wie man beispielsweise daran erkennen kann, dass Leerräume vom Compiler aus dem Text entfernt werden), aber logisch wird er dazugezählt. Ein Leerraum ist ein Zeichen, das an jedem Ort zwischen allen Token mit Funktion platziert werden kann. Eine beliebige Anzahl kann hintereinander notiert werden. Leerräume haben keinerlei andere Bedeutung, als den Quellcode übersichtlich zu gestalten. Java erkennt die folgenden Zeichen als Leerraum an: • [Space] • [Tab] • [Zeilenende] • [Formularvorschub]

Warnungen

- Java benutzt keinen 8-Bit-ASCII-Code, sondern den hexadezimal verschlüsselten 16-Bit-Unicode-Zeichensatz. Die ersten 256 Zeichen entsprechen dem normalen ASCII-Zeichensatz (Byte eins ist auf 0 gesetzt). Eine ASCII-Kodierung wird einfach durch eine kanonische Übersetzung (das bedeutet, die Reihenfolge und Anordnung der ASCII-Kodierung sind auch in der neuen Kodierung als Block wiederzufinden) mit Voranstellen der Zeichenfolge `\u00` und folgender Hexadezimalzahl in das dazu passende Unicode-Zeichen übersetzt. Dies definiert in Java eine Sequenz, mit der alle Unicode-Zeichen verschlüsselt werden können. Alle vorkommenden Zeichen in Java, wie Buchstaben, Zahlen, Sonderzeichen usw., bestehen aus Unicode. Der Java-Compiler erwartet auf jeden Fall Unicode. Gegebenenfalls transformiert er gewöhnlichen ASCII-Code vor einer Kompilierung automatisch. Zwei Token gelten in Java nur dann als identisch, wenn ihre Unicode-Darstellung übereinstimmend ist – sowohl dieselbe Länge, als auch vollkommende Übereinstimmung jedes Zeichens bezüglich des Unicode-Wertes. Dies ist auch der Grund, warum in Java unbedingt zwischen Groß- und Kleinschreibung zu unterscheiden ist.
- Sämtliche Schlüsselworte haben in Java eine spezifische Bedeutung und können also nicht als Bezeichner für Variablen, Konstanten, Klassennamen usw. benutzt werden.
- Da Java Unicode-Zeichenliterale verwendet, muss man bei Zeichenliteralen aufpassen, wenn Escape-Unicode-Literale verwendet werden. Unicode-Zeichenliterale werden schon zu einem sehr frühen Zeitpunkt vom Java-Compiler interpretiert. Wenn man daher die Escape-Unicode-Literale dazu verwendet, ein zeilenbeendendes Zeichen, wie beispielsweise Wagenrücklauf oder neue Zeile, darzustellen, wird das Zeilenendezeichen vor dem schließenden einfachen Anführungszeichen interpretiert. Das Resultat ist dann ein Kompilierfehler. Das `\u`-Format ist also ungeeignet, um ein Zeilenendezeichen darzustellen. Die Thematik ist sogar noch allgemeiner zu verstehen. Bei Zeichenkettenlitera-

len müssen immer beide Anführungszeichen in derselben Zeile des Quellcodes stehen, damit die Zeichenkette nicht aus Sicht des Compilers automatisch ein Zeichen für eine neue Zeile enthält. Wenn innerhalb eines Zeichenkettenliterals eine neue Zeile notiert werden soll, muss eine Escape-Sequenz wie beispielsweise `\n` oder `\r` benutzt werden. Die Zeichen für doppelte Anführungszeichen (") und Backslash (\) müssen ebenfalls durch die Verwendung von Escape-Sequenzen (`\"` und `\\`) dargestellt werden.

Beispiele

Bezeichner:

```
HelloWorld
```

Schlüsselwort:

```
void
```

Zeichenkettenliteral:

```
"Jetzt ein Tab\tund danach eine neue Zeile\n"
```

Ganzzahlliterale:

Dezimal	Hexadezimal	Oktal
42	0X2A	052
1111	0X457	02127

Gleitzahlliterale:

```
float gleitzahl = 42.123F;
float gleitzahl = 0.1f;
double gleitzahl = 0.123;
```

Anwendung von arithmetischen Operatoren:

```
ergebnis = a + b;
ergebnis = ergebnis / 42;
ergebnis = 11 % 5;
```

Bitweiser And-Operator[6]:

```
00101010 & 00101010 = 00101010
```

Bitweiser OR-Operator:

```
11100011 | 11101100 = 11101111
```

Bitweiser XOR-Operator:

```
11110000 ^ 00001111 = 11111111
```

6. **Die nachfolgenden bitweisen Beispiele sollen keine vollständigen Codebeispiele sein, sondern nur die Wirkungsweise dieser Operatoren zeigen.**

Bitweiser Komplement-Operator:

```
~ 00001000 = 11110111
```

Bitweise Verschiebungsoperatoren:

```
00111100 << 2= 11110000
01001111 >> 1= 00100111
01001111 >>> 1= 00100111
```

Überladen

Eine Realisierung des polymorphen Verhaltens von Java mittels einer allgemeinen OO-Technik (OO = Objektorientierung). Polymorphes Verhalten bedeutet, in Java können verschiedene Operationen in der OOP den gleichen Namen haben. Auch wenn sie auf verschiedene Objekte angewandt werden, was in der prozedualen Welt mehrere unterschiedliche (auch namentliche) Operationen (dort Funktionen oder Prozeduren) zur Folge hätte.

Beschreibung

Java erlaubt ein Überladen (Overloading) von Methoden (jedoch kein Überladen von Operatoren). Das Methoden-Overloading bedeutet die konkrete Realisierung des polymorphen Verhaltens von Java.

Anwendung

Überladen bedeutet, dass bei Namensgleichheit von Methoden in Superklasse und abgeleiteter Klasse die Methoden der abgeleiteten Klasse die der Superklasse überdecken. Dabei wird zwingend vorausgesetzt, dass sich die Parameter signifikant (entweder in der Anzahl und/oder dem Typ) unterscheiden, sonst handelt es sich bei der Konstellation von namensgleichen Methoden in Super- und Subklasse um den Vorgang des Überschreibens.

Bei Methoden der gleichen Klasse kann es analog den Vorgang des Überladens geben (das findet man beispielsweise oft bei Kontruktormethoden). Im Gegensatz zum obigen Fall einer abgeleiteten Klasse können Methoden derselben Klasse sich nicht gegenseitig überschreiben, das bedeutet, ihre Signatur muss sich signifikant unterscheiden.

Warnungen

- Methoden, die mit dem Schlüsselwort `static` deklariert werden, können nicht überladen werden in dem Sinn, dass in derselben Klasse sich zwei Methoden nur durch dieses Schlüsselwort unterscheiden.
- Es genügt zum Überladen einer Methode nicht, wenn sich nur der Rückgabetyp der neuen Methode von dem der Ursprungsmethode unterscheidet. Auch verschiedene Namen von Parametern sind nicht signifikant.

Beispiel

Die Methode `meineMethode(int i)` wird in der abgeleiteten Klasse überladen.

```
class SuperKlasse
{
 public static void meineMethode(int i)
 {
...
 }
}
class ueberlade extends SuperKlasse
{
 public static void meineMethode(int i, int j)
 {
...
 }
 public static void main (String args[])
 {
...
  meineMethode(2,2); //Aufruf der Methode in der Subklasse
/* Aufruf der Methode in der Superklasse */
  meineMethode(1);
...
 }
}
```

Überschreiben

Eine allgemeine OO-Technik zur Spezialisierung von Klassen beziehungsweise Objekten.

Beschreibung

Überschreiben (engl. Overriding, was eigentlich Überdefinieren bedeutet) ist dem Mechanismus des Überladens sehr ähnlich. Es bedeutet, dass in einer abgeleiteten Klasse eine von der Superklasse geerbte Methode neu definiert wird. Im Gegensatz zum Überladen hat die neue Methode dabei den gleichen Namen und die gleiche Signatur (also keine signifikante Unterscheidung).

Anwendung

Das Erstellen zweier Methoden mit gleichem Namen und Parametersignaturen innerhalb derselben Klasse (des gleichen Namensraums) ist nicht gestattet. Innerhalb einer Subklasse kann allerdings eine Methode gleichen Namens wie der Name einer Methode in der Superklasse erstellt werden. Diese wird die Methode der Superklasse überschreiben.

Wichtig ist, dass sich die Parametersignaturen – im Gegensatz zum Überladen – nicht signifikant (also weder in Anzahl, noch im Typ) unterscheiden. Auch darf sich der Rückgabewert nicht unterscheiden. Die Namen der Parameter spielen jedoch keine Rolle! Die Modifier sind nicht relevant, dürfen aber bei einer Abweichung in der Subklasse die Sichtbarkeit der Methode maximal einschränken.

Beispiele

Die Methode der Superklasse wird in der Subklasse überschrieben:

```
class SuperKlasse
{
...
 public static void meineMethode(int i) //Methode der Superklasse
 {
...
 }
}
class ueberschreibe extends SuperKlasse
{/* gleicher Methodenname in der Subklasse */
//Identische Methodendeklaration
 public static void meineMethode(int i)
 {
...
 }
```

```
 public static void main (String args[])
 {
...
// Aufruf der Methode der abgeleiteten Klasse
  meineMethode(42);
...
 }
}
```

Variablen

Benannte Stellen im Hauptspeicher.

Beschreibung

Variablen bezeichnen Stellen im Hauptspeicher, in denen irgendwelche Werte temporär gespeichert werden können. Dazu haben sie einen Namen und einen spezifischen Datentyp.

Anwendung

Java kennt drei unterschiedliche Arten von Variablen, deren Bezeichner den üblichen Regeln für Token folgen:

- Instanzvariablen
- Klassenvariablen
- Lokale Variablen

Instanzvariablen beziehungsweise Objektvariablen werden zum Definieren von Attributen oder eines Zustandes eines bestimmten Objektes benutzt. Dazu werden Instanzvariablen innerhalb der Klasse definiert und stehen in allen Instanzen der Klasse zur Verfügung. Die dortige Veränderung des Wertes der Variablen hat keine Auswirkungen auf den Wert einer Variablenn in einer anderen Instanz. Angesprochen werden sie über die Punktnotation. Dabei steht bei Bedarf auf der linken Seite des Punktes das Objekt, zu der die Variable gehört, und auf der rechten Seite der Name der Variablen.

Auf beiden Seiten des Punktes steht ein Ausdruck (im Java-Sinn). Damit können Instanzvariablen auch verschachtelt werden. Wenn die Instanzvariable selbst ein Objekt beinhaltet und dieses Objekt wiederum eine eigene Instanzvariable, so kann darauf über die Punktnotation Bezug genommen werden. Die Punktnotation wird dabei von links nach rechts bewertet.

Klassenvariablen sind Instanzvariablen sehr ähnlich. Sie werden wie Instanzvariablen in der Klasse deklariert. Nur muss der Deklaration das Schlüsselwort `static` vorangestellt werden. Klassenvariablen werden nur einmal physikalisch im Hauptspeicher angelegt. Jede Instanz der Klasse kann darauf zugreifen und die Werte der Variablen ändern (sofern nicht als final deklariert). Dies bedeutet, dass eine Änderung, die von einer bestimmten Instanz der Klasse durchgeführt worden ist, für alle anderen Instanzen ebenfalls sichtbar ist. Der Zugriff auf Klassenvariablen erfolgt wieder mit der Punktnotation. Dabei steht bei Bedarf bei einem Aufruf von außen vor dem Punkt der Name der Klasse und nicht der einer Instanz.

Lokale Variablen werden innerhalb von Methodendefinitionen deklariert und können auch nur dort benutzt werden. Dies können Indexvariablen von Schleifen (sogenannte schleifenlokale Variablen) sein oder temporäre Variablen zur Auf-

nahme von Werten, welche nur innerhalb der Methodendefinitionen Sinn machen. Es gibt gleichfalls auf Blöcke beschränkte lokale Variablen. Schleifenlokale Variablen (etwa Zählvariablen in `for`-Schleifen) müssen nicht unbedingt außerhalb der Schleife (jedoch noch innerhalb der Methode) vereinbart werden, sondern können ebenso direkt im Initialisierungsteil der Schleife direkt vereinbart werden und sind dann auch nur dort bekannt. Diese im Initialisierungsteil einer Schleife direkt vereinbarten Zählvariablen sind ein Sonderfall von normalen lokalen Variablen und überdecken gegebenenfalls gleichnamige Variablen im übergeordneten Programmblock (auch der Methode selbst, welche die Schleife beinhaltet). Die Variablen im übergeordneten Programmblock bleiben hierdurch unverändert.

Lokale Variablen existieren nur solange im Speicher, wie die Methode, die Schleife oder der Block existieren. Wesentlichster Unterschied von lokalen Variablen zu Instanz- und Klassenvariablen ist, dass sie unbedingt einen Wert zugewiesen bekommen müssen, bevor sie benutzt werden können. Instanz- und Klassenvariablen haben dagegen einen typspezifischen Defaultwert. Es gibt einen Compilerfehler, wenn lokale Variablen nicht vor der ersten Verwendung initialisiert wurden.

Ein weiterer Unterschied von lokalen Variablen zu Instanz- und Klassenvariablen ist, dass man auf sie nur direkt im lokalen Bereich und nicht von außen über die Punktnotation zugreifen kann.

Durch Voranstellen von `final` wird aus einer Klassen- oder Instanzvariablen eine Konstante. Bei einer solchen Deklaration muss logischerweise ein Anfangswert gleich mitzugewiesen werden.

Lokale Variablen können hingegen nicht final deklariert werden.

Beispiele

Instanzvariablen:

```
public class MeineKlasse
{
int meineVar;
String str;
...
}
```

Zugriff auf eine Instanzvariable:

```
MeinObjekt.meineVar;
```

Verschachtelte Instanzvariablen, wo zuerst die Instanzvariable des links stehenden Objektes bewertet wird. Diese zeigt auf ein anderes Objekt mit der rechts stehenden Variable, deren Wert zurückgegeben wird:

```
MeinObj.MeinObj2.enthalteneInstanzVar;
```

Klassenvariablen:

```
public class MeineKlasse
{
static int zaehler;
static String str;
...
}
```

Konstanten:

```
final int antwort = 42;
final String bye = "Und Tschüss";
```

Die Java-Schlüsselworte

Jede Programmiersprache enthält eine gewisse Anzahl von Buchstabenfolgen, die neben der Sprachgrammatik der wesentliche Teil der Sprachdefinition sind. Diese sogenannten Schlüsselworte haben eine definierte Bedeutung (sie bilden damit ein sogenanntes Token – Sinnzusammenhang – mit besonderer Bedeutung für sämtliche Sprachwerkzeuge wie dem Compiler) und bilden den Kern einer Programmiersprache, die einem Programmier auf jeden Fall bekannt sein muss.

Schlüsselworte haben also eine spezifische Bedeutung und können nicht als Bezeichner für Variablen, Konstanten, Klassennamen usw. benutzt werden. Da Java Groß- und Kleinschreibung unterscheidet, gilt dies natürlich auch für Schlüsselworte. Groß- und Kleinschreibung ist zu unterscheiden. Sämtliche Schlüsselworte in Java werden klein geschrieben. Theoretisch ist es also erlaubt, Token zu erzeugen, die sich von Schlüsselworten nur durch einen oder mehrere Großbuchstaben unterscheiden. Davon ist jedoch dringend(!) abzuraten. Die Schlüsselworte von Java 2 lauten wie folgt.

Reservierte Java-Schlüsselworte:

abstract	boolean	break	byte	case
catch	char	class	const	continue
default	do	double	else	extends
false	final	finally	float	for
goto	if	implements	import	instanceof
int	interface	long	native	new
null	package	private	protected	public
return	short	static	super	switch
synchronized	this	throw	throws	transient
true	try	void	volatile	while

Einige Token wurden Java von Anfang an reserviert, obwohl sie noch keine definierte Bedeutung hatten. Dies sind die Token

- byvalue
- cast
- const
- future
- generic
- goto
- inner

- `operator`
- `outer`
- `rest`
- `var`

Bei diesen Token werden nur `const` und `goto` offiziell zu den Schlüsselworten von Java gezählt (obwohl derzeit ohne Funktionalität). Alle anderen Token in dieser Aufzählung haben in der Java-Version 2 keine weitergehende Bedeutung, als dass sie gegen eine Verwendung als Bezeichner geschützt sind.

Dieser Schutz macht jedoch Sinn und einem Token kann in einer folgenden Java-Version eine Bedeutung zukommen, wie das Schlüsselwort `transient` zeigt. Es hatte in bisherigen Java-Versionen noch keine Bedeutung, aber mittlerweile eine wohldefinierte Funktionalität.

Die Details

In dem Kapitel behandeln wir alphabetisch aufgelistet die einzelnen Schlüsselworte. Dabei verfolgen wir bei jedem besprochenen Schlüsselwort folgenden Aufbau:

Kategorie	Was steht da	Wann vorhanden
Die Titelzeile	Der Name des Schlüsselwortes, welches beschrieben werden soll.	Immer vorhanden
Die Beschreibung	Eine Kurzbeschreibung der Bedeutung des Schlüsselwortes.	Immer vorhanden
Die Anwendung	Die eigentliche Referenz. Hier erfolgt die Beschreibung der Aufgabe des Schlüsselwortes.	Immer vorhanden
Warnungen	Mögliche Gefahrenpotentiale des Schlüsselwortes.	Optional. Dieser Abschnitt ist immer dann vorhanden, wenn eine oder mehrere Warnung(en) angebracht beziehungsweise sinnvoll ist (sind).
Beispiele	Kommentierte Beispiele, welche die Anwendung des Schlüsselwortes erläutern.	Immer vorhanden
Tips	Unter Umständen einige weitere und abschließende Tips.	Optional
Verwandte Befehle	Verweise auf Schlüsselworte innerhalb des Buchs, die in sinnvollem Zusammenhang zu dem besprochenen Schlüsselwort stehen.	Optional, aber fast immer vorhanden.

Hinweis: Der in verwandten Titeln der Buchreihe besprochene Abschnitt über Parameter, die für den Befehl sinnvoll oder notwendig sind, macht bei einem Aufbau dieses Kapitels über Java-Schlüsselworte keinen Sinn. Schlüsselworte sind meist keine vollständigen Befehle (wenngleich zentraler Bestandteil einer Befehlsstruktur) und sie haben dementsprechend keine Parameter.

abstract

Ein Modifier für Klassen, Schnittstellen und Methoden.

Beschreibung

Dieses Schlüsselwort dient als Modifier, um Klassen, Schnittstellen und Methoden als abstrakt zu kennzeichnen. Er wird der Deklaration einfach vorangestellt.

Anwendung

Unter einer abstrakten Klasse versteht man eine Klasse, von der nie eine direkte Instanz benötigt wird und von welcher auch keine Instanz unmittelbar gebildet werden kann. Sie dient meist zur Festlegung von einem oder mehreren allgemeinen Verweis(en) beziehungsweise der Zusammenfassung von gemeinsamem Code nachfolgender Subklassen.

Bei abstrakten Methoden handelt es sich um Methoden, die zwar deklariert sind, aber noch nicht in die aktuelle Klasse implementiert wurden. Daher muss eine abstrakte Methode in der Subklasse, welche die implementierende Klasse als Superklasse hat, überschrieben und implementiert werden, bevor mit der Methode irgendetwas anzufangen ist. In einer als abstrakt deklarierten Methode ersetzen Sie den Body (Körper) der Methode inklusive der geschweiften Klammern durch ein einfaches Semikolon (;).

Im Falle von Schnittstellen ist der Modifier bei Methoden nicht explizit notwendig, denn alle Schnittstellen sind abstrakt. Der Modifier kann jedoch trotzdem aufgrund der Eindeutigkeit gesetzt werden.

Prinzipiell dient abstrakter Java-Quellcode zur Deklaration von noch nicht vollständigem Code. Wenn dies ohne die Kennzeichnung als `abstract` erfolgt, wird durch den Compiler ein Fehler erzeugt. Die Deklaration von Quellcode als abstrakt bedeutet auf der anderen Seite jedoch nicht, dass vollständiger Code einen Fehler erzeugt. Auch muss eine als abstrakt gekennzeichnete Quellcodestruktur keinen unvollständigen Code enthalten. Es ist nur wenig sinnvoll, solchen Code als abstrakt zu kennzeichnen.

Warnungen

- Die Schlüsselwörter `final` und `abstract` können nicht zusammen bei einer Klasse verwendet werden. Abstrakte Methoden dürfen ebenfalls nie als `final` deklariert werden, weil dadurch verhindert wird, dass diese Methoden überschrieben werden können. Sie wären nutzlos.
- Weder statische Methoden noch Klassenkonstruktoren dürfen als abstrakt deklariert werden.
- Sofern eine Subklasse nicht alle abstrakten Methoden überschreibt, muss auch sie als abstrakt deklariert werden.

Beispiele

Eine abstrakte Klasse:

```
abstract class MeineKlasse
```

Eine abstrakte Methode:

```
abstract boolean sonstnix();
```

Verwandte Befehle

`final`
`public`

boolean

Ein logischer Datentyp.

Beschreibung

Werte dieses Datentyps können die Werte `true` (wahr) oder `false` (falsch) annehmen.

Anwendung

Ein boolescher Datentyp wird im Wesentlichen bei Vergleichen in Kontrollstrukturen benötigt. Alle logischen Vergleiche in Java liefern einen Rückgabenwert vom Typ `boolean`.

Warnungen

Auch boolesche Variablen sind dem eigenen Java-Typ `boolean` zugeordnet und können nur die Werte `true` oder `false` annehmen. Die Zuordnung eines booleschen Datentyps mit der Zahl Null oder Ungleich Null (was beispielsweise in der verwandten Sprache C/C++ erlaubt ist) erzeugt einen Fehler durch den Compiler. Zahlenoperationen können mit dem booleschen Datentyp explizit nicht durchgeführt werden. Werte vom Typ `boolean` sind zu allen anderen primitiven Datentypen inkompatibel und lassen sich nicht direkt durch Casting in andere Typen überführen.

Beispiele

Eine Methodendeklaration mit einem booleschen Rückgabewert:

```
boolean jaodernein();
```

Eine boolesche Variable:

```
boolean test;
```

Eine boolesche Variable mit Zuweisung:

```
boolean test=false;
```

Eine Kontrollstruktur mit direkter Auswertung der booleschen Variablen:

```
if (test) System.out.println("Hallo");
```

Eine Kontrollstruktur mit Auswertung des Vergleichs:

```
if (testvariable==0) System.out.println("Hallo");
```

Verwandte Befehle

```
byte
char
double
float
int
long
short
false
true
```

break

Eine Sprunganweisung.

Beschreibung

Diese Sprunganweisung dient dazu, Unteranweisungsblöcke von Schleifen und `switch`-Anweisungen zu verlassen. Es gibt sowohl bezeichnete, als auch unbezeichnete `break`-Anweisungen.

Anwendung

Eine unbezeichnete `break`-Anweisung wird im Quellcode notiert (oft innerhalb einer Entscheidungsstruktur) und springt zu der nächsten Zeile hinter der derzeitigen (innersten) Wiederholungs- oder `switch`-Anweisung.

Mit einer bezeichneten `break`-Anweisung kann an eine Anweisung in der derzeitigen Methode gesprungen werden, die mit dieser Bezeichnung gekennzeichnet ist. Für eine solche Kennzeichnung muss vor dem Anfangsteil der zu unterbrechenden (Unter-) Struktur ein Label (eine Beschriftung) eingegeben werden. Dies erfolgt mit einem Namen, gefolgt von einem nachgestellten Doppelpunkt. Sofern ein falsch gesetztes oder nicht vorhandenes Label in einer Sprunganweisung angegeben wird, wird bereits der Compiler eine Fehlermeldung ausgeben.

Wenn es in Verbindung mit der `break`-Anweisung einen `finally`-Teil einer derzeit aktiven `try`-Anweisung gibt, wird immer dieser Codeabschnitt zuerst ausgeführt, bevor die Kontrolle weitergegeben wird.

Warnungen

Interessant sind benannte Schleifen besonders in Verbindung mit verschachtelten Schleifen. Damit kann durchaus ein "Spagetticode" analog den früheren "`goto-return`"-Konstrukten erzeugt werden. Die Situation wird jedoch dadurch entschärft, dass nur Sprünge über mehrere Blöcke innerhalb einer Schleifen-/Auswahlstruktur durchgeführt werden können.

Beispiele

Unbezeichnete `break`-Anweisungen:

```
int test; // Eine numerische Testvariable
...
switch (test)
  {
  case 0:
   ...
   break;
  case 1:
   ...
   break;
  case 2:
   ...
   break;
  case 3:
   ...
   break;
}
...
```

Ein Beispiel mit unbezeichneten und bezeichneten `break`-Anweisungen. Das Label ist der Struktur vorangestellt. Dorthin springt die bezeichnete `break`-Anweisung:

```
sprungziel:
switch (test)
  {
case 0:
      {
      ...
      break;
      }
case 1:
      {
      ...
      break;
      }
case 2:
      {
      ...
      break sprungziel; //springt zum Label
      }
...
}
```

Eine bezeichnete `break`-Anweisung, welche über zwei Ebenen springt (in einer verschachtelten Struktur) :

```
sprungziel:
switch (test)
{
case 2:
      {
      switch (test2)
      {
      ...
       case 4:
        {
  ...
         break sprungziel; //springt zum Label
       }
...
      }
...
}
```

Tips

Die Anweisung `break` kann gezielt weggelassen werden, um damit spezielle Effekte zu erzielen. Beispielsweise werden dann innerhalb einer `switch-case`-Konstruktion nach einem Treffer alle weiteren `case`-Anweisungen ausgeführt.

Verwandte Befehle

```
continue
return
throw
```

byte

Ein Datentyp (Ganzzahltyp).

Beschreibung

Der Datentyp `byte` ist mit einer Länge von 8 Bit der kleinste Wertebereich von Java. Defaultwert ist 0. Er hat ein Vorzeichen. Dieser Datentyp kann die Ganzzahlwerte (ganzzahliges Zweierkomplement) von (- 2 hoch 7 =) -128 bis (+ 2 hoch 7 – 1 =) +127) annehmen.

Anwendung

Der Datentyp kann in Java vielseitig eingesetzt werden. So ist er, um ein Beispiel zu nennen, gültiger Datentyp in `switch`-Anweisungen und für Ein- und Ausgabe mit diesem Typ stehen besondere Methoden in Java zur Verfügung. Eine genauere Beschreibung der potentiellen Anwendungen sprengt den Rahmen. Prinzipiell wird mit diesem Schlüsselwort einer Variablen ein Datentyp oder einer Methode ein Rückgabetyp zugewiesen.

Warnungen

Bei Operationen mit zwei `byte`-Datentypen entsteht immer nur ein Ergebnis vom Datentyp `int` oder `long`.

Beispiele

Eine Methodendeklaration mit einem Rückgabewert vom Typ `byte`:

```
byte test();
```

Eine Variable vom Typ `byte`:

```
byte test;
```

Verwandte Befehle

Die übrigen Datentypen von Java:

```
boolean
char
double
float
int
long
short
```

case

Eine Benennung einer auszuwählenden Bezeichnungskonstante. Ein Bestandteil der `switch-case-default`-Auswahlanweisung.

Beschreibung

Über eine Auswahlanweisung wird einer von mehreren möglichen Kontrollflüssen ausgesucht.

Anwendung

Die `case`-Anweisung kann nur in Verbindung mit der einleitenden `switch`-Anweisung verwendet werden. Diese ermöglicht das Weitergeben des Kontrollflusses an eine von vielen Anweisungen in Ihrem Block mit Unteranweisungen. Diese werden dann mit `case` eingeleitet.

An welche `case`-Anweisung innerhalb der `switch`-Anweisung der Kontrollfluss weitergereicht wird, hängt vom Wert des Ausdrucks in der Anweisung ab. Es wird die erste Anweisung nach einer `case`-Bezeichnung ausgeführt, welche denselben Wert wie der in der `switch`-Anweisung getestete Ausdruck hat. Wenn es keine entsprechenden Werte gibt, wird die erste Anweisung hinter der `default`-Bezeichnung ausgeführt. Wenn auch die nicht vorhanden ist, wird die erste Anweisung nach dem `switch`-Block ausgeführt.

Bezeichnungen beeinflussen den Kontrollfluss nicht. Die Kontrolle behandelt diese Bezeichnungen so, als wenn sie nicht vorhanden wären. Daher können beliebig viele Bezeichnungen vor derselben Codezeile stehen.

Über eine `break`-Anweisung kann ein ausgewählter Block verlassen werden.

Warnungen

- Die zu testenden `switch`-Ausdrücke und `case`-Bezeichnungskonstanten müssen alle vom Typ `byte, short, char` oder `int` sein.
- Es dürfen keine zwei oder mehr `case`-Bezeichnungen im gleichen Block denselben Wert haben.
- Der Kontrollfluss wird nach einer Übereinstimmung mit einer `case`-Bezeichnung und nachfolgender Abarbeitung nicht automatisch aus der `switch`-Struktur zurückgegeben. Alle nachfolgenden `case`-Blöcke werden ebenfalls abgearbeitet! Dies kann man mit einer `break`-Anweisung am Ende eines `case`-Blocks unterbinden.

Beispiele

Eine Auswahl über eine numerische Testvariable:

```
int test; // Eine numerische Testvariable
...
switch (test)
  {
  case 0:
   ...
   break;
  case 1:
   ...
   break;
  case 2:
   ...
   break;
  case 3:
   ...
   break;
}
...
```

Tips

Um den Kontrollfluss nach einer Übereinstimmung mit einer `case`-Bezeichnung und nachfolgender Abarbeitung aus der `switch`-Struktur ohne Abarbeitung der nachfolgenden `case`-Blöcke zurückzugeben, muss eine `break`-Anweisung am Ende eines `case`-Blocks folgen. Natürlich können Sie dieses Verhalten auch gezielt einsetzen, um durch Auslassen von `break`-Anweisungen den Durchlauf von mehreren Blöcken zu erreichen.

Verwandte Befehle

`switch`
`break`
`default`
`if`
`if-else`

catch

Das Schlüsselwort ist zentraler Teil der Schutzanweisung zum Auffangen von Ausnahmen in Java.

Beschreibung

Java verfolgt zum Abfangen von Laufzeitfehlern ein Konzept, welches mit sogenannten Ausnahmen arbeitet. Ausnahmen werden unter Java über ein `try-catch-finally`-Konstrukt aufgefangen. Der `finally`-Zweig ist optional. Dabei wird der mit `catch` eingeleitete Block zum Deklarieren von den Schritten beim Auftreten einer Ausnahme verwendet. Das Schlüsselwort kann nur in Zusammenhang mit einem vorangestellten `try`-Block eingesetzt werden.

Anwendung

Während der Laufzeit eines Programms kann es nichtplanbare Situationen geben, welche zur Laufzeit eines Programms auftreten und eine unmittelbare Reaktion durch das Programm beziehungsweise den Anwender notwendig machen. Etwa, wenn ein Zugriff auf ein Diskettenlaufwerk erfolgt, wo keine Diskette eingelegt ist, oder die Division durch Null während einer mathematischen Operation. Ein solcher Vorgang wird in Java eine Ausnahme erzeugen, welche dann unmittelbar vom Programm bearbeitet werden muss.

Damit diese Ausnahmen sicher gehandhabt werden können, stellt Java Schutzanweisungen zur Verfügung. Diese Anweisungen benutzen im Wesentlichen die drei Schlüsselworte `try`, `catch` und `finally` in Kombination.

Innerhalb des mit `try` eingeleiteten Blocks werden die Anweisungen notiert, welche eine Ausnahme erzeugen können. Kritische Aktionen in einem Java-Programm sollten immer innerhalb eines `try`-Blocks durchgeführt werden.

Innerhalb des mit `catch` eingeleiteten Blocks werden die Anweisungen notiert, welche beim Auftreten einer bestimmten Ausnahme durchgeführt werden sollen. Wenn also eine der Anweisungen innerhalb des `try`-Blocks ein Problem erzeugt, wird dieses durch die passende `catch`-Anweisung aufgefangen und entsprechend behandelt (sofern die `catch`-Anweisung dafür die passende Behandlung enthält – `catch`-Anweisungen haben Ausnahmeobjekte als Parameter). Am Ende eines `try`-Blocks können beliebig viele `catch`-Klauseln stehen. Sie werden einfach nacheinander notiert. Damit können unterschiedliche Arten von Ausnahmen auch verschiedenartig – und damit sehr qualifiziert – gehandhabt werden.

Die optionale `finally`-Anweisung erlaubt die Abwicklung wichtiger Abläufe (wie beispielsweise das Schließen von Dateien), bevor die Ausführung des gesamten `try-catch-finally`-Konstruktes beendet wird. Der in dem `finally`-Block untergebrachte Code wird in jedem Fall abgearbeitet (ohne oder mit Auftreten einer Ausnahme).

Beispiele

Ein einfaches `try-catch`-Konstrukt mit einer `catch`-Klausel:

```
try
{
//  kritische Anwendungen
}
catch (Exception e)
{
// Behandlung der Ausnahme
// Das Ausnahmeobjekt e wird behandelt
}
```

Ein einfaches `try-catch`-Konstrukt mit einer `catch`- und einer `finally`-Klausel:

```
try
{
//  kritische Anwendungen
}
catch (Exception e)
{
// Behandlung der Ausnahme
// Das Ausnahmeobjekt e wird behandelt
}
finally
{
// Abschlußarbeiten
}
```

Ein `try-catch`-Konstrukt mit mehreren `catch`-Klauseln:

```
try
{
//  kritische Anwendungen
}
catch (MeineException1 e)
{
// Behandlung der Ausnahme 2 - selbstdefiniert
}
catch (MeineException2 e)
{
// Behandlung der Ausnahme 3 - selbstdefiniert
}
catch (Exception e)
{
// Behandlung der Ausnahme 1
// Das Ausnahmeobjekt e wird behandelt
}
```

Verwandte Befehle

try
finally
switch
break
default
if
if-else

char

Ein Datentyp zur Darstellung von Zeichen.

Beschreibung

Der Datentyp dient zur Darstellung eines Zeichens des unter Java verwendeten Unicode-Zeichensatzes. Ein Element vom Typ `char` hat eine Länge von 16 Bit. Zur Darstellung von alphanumerischen Zeichen wird dieselbe Kodierung wie beim ASCII-Zeichensatz verwendet, aber das höhere Byte ist auf `0` gesetzt. Defaultwert ist `\u0000`. Der Maximalwert, welchen der Datentyp `char` darstellen kann, ist `\uFFFF`. Der Datentyp ist als einziger primitiver Java-Datentyp vorzeichenlos!

Anwendung

Der Datentyp kann in Java so vielseitig eingesetzt werden, dass eine genaue Beschreibung der potentiellen Anwendungen den Rahmen sprengt. Insbesondere werden aus `char`-Elementen Zeichenketten (Strings) zusammengesetzt. Auch bitweise Arithmetik ist für den Datentyp definiert. Ebenso können die Werte zweier Variablen vom Typ `char` mittels Vergleichsoperatoren verglichen werden. Dabei werden die Variablen vom Typ `char` bei der Verwendung eines Vergleichsoperators (oder auch bei anderen Operationen) wie ganze 16-Bit-Zahlen (Werte 0 bis 65535) entsprechend ihrer Unicode-Kodierung behandelt. Auch eine Verknüpfung von Variablen vom Typ `char` mit anderen Variablen gleichen Typs oder auch anderen Datentypen ist erlaubt. Dabei findet unter Umständen eine automatische Typumwandlung statt. Generell wird mit diesem Schlüsselwort einer Variablen ein Datentyp oder einer Methode ein Rückgabetyp zugewiesen.

Warnungen

Recht offensichtlich ist die Tatsache, dass unter Umständen einige Informationen verloren gehen, wenn `char`-Zeichen in einen kleineren Typ (Länge ein Byte) umgewandelt werden. Aber auch die Umwandlung von einer Variablen vom Datentyp `char` in eine Variable vom Datentyp `short` (Länge zwei Byte) kann zu Informationsverlust führen – trotz gleicher Länge der Datentypen. Der Grund ist die Darstellung von `short` als ganzzahliges Zweierkomplement mit Vorzeichen, während `char` vorzeichenlos wie ganze 16-Bit-Zahlen (Werte 0 bis 65535) entsprechend der Unicode-Kodierung behandelt wird.

Beispiele

Eine Methodendeklaration mit einem Rückgabewert vom Typ `char`:

```
char test();
```

Eine Variable vom Typ `char`:

```
char test;
```

Eine Verknüpfung einer Variablen vom Typ `char` und einer Variablenn vom Typ `int`:

```
char test = 'z';
int wert = 5;
System.out.println(test + wert);
```

Verwandte Befehle

Die übrigen Datentypen von Java:

```
boolean
byte
double
float
int
long
short
```

class

Der Beginn einer Klassendeklaration.

Beschreibung

Mit diesem Schlüsselwort wird festgelegt, dass es sich bei dem nachfolgenden Quelltext um die Deklaration einer Klasse handelt. Generell haben Klassendeklarationen in Java folgendes Format:

```
[modifiers] class NeueKlasse [extends NamederSuperKlasse] [implements
NamederSchnittstelle(n)]
```

wobei alles in eckigen Klammern optional ist. Dabei kann auch mehr als nur eine Schnittstelle implementiert werden. Dann müssen mehrere Schnittstellen mit der `implements`-Anweisungen hintereinander geschrieben werden (durch Kommata getrennt).

Anwendung

In Verbindung mit den vier weiteren Angaben über eine Klasse

- dem Modifier
- dem Klassennamen
- der Superklasse
- den Schnittstellen

deklariert `class` eine Klasse.

Beispiele

Eine minimale Klassendeklaration:

```
class MeineKlasse
  {...}
```

Eine Klassendeklaration, welche eine Superklasse erweitert:

```
class MeineKlasse extends meineSuperklasse
  {...}
```

Eine Klassendeklaration mit Modifier und einer implementierenden Schnittstelle:

```
public class MeineKlasse implements MeineSchnittstelle
  {...}
```

Eine Klassendeklaration mit zwei implementierenden Schnittstellen:

```
class MeineKlasse implements MeineSchnittstelle, MeineZweiteSchnittstelle
  {...}
```

Verwandte Befehle

`interface`
`package`

const

Ein reserviertes Schlüsselwort, welches in dem aktuellen API ohne Funktionalität ist.

Beschreibung

Es gibt in Java einige Token, die vorsorglich für spätere Versionen von Java reserviert wurden, sonst aber derzeit noch keine weitere Bedeutung haben. Es handelt sich um die Token `byvalue`, `cast`, `const`, `future`, `generic`, `goto`, `inner`, `operator`, `outer`, `rest` und `var`. Von diesen Token werden aber nur `const` und `goto` offiziell zu den Schlüsselworten von Java gezählt.

Anwendung

Derzeit ohne konkrete Anwendung.

Verwandte Befehle

Die übrigen reservierte Token ohne konkrete Funktionalität (verwandt im Sinne von »derzeit ohne konkrete Funktionalität«).

continue

Eine Sprung- beziehungsweise Abbruchanweisung.

Beschreibung

Diese Anweisung dient dazu, einen aktuellen Schleifendurchlauf zu unterbrechen.

Anwendung

Durch die Anweisung `continue` wird im Gegensatz zu der verwandten Anweisung `break` nicht die gesamte Schleife abgebrochen, sondern nach Abbruch des aktuellen Schleifendurchlaufs der nächste Durchlauf einer Schleife durchgeführt. Dabei ist wie bei `break` zwischen benannten und unbenannten Spüngen zu unterscheiden.

Falls hinter `continue` kein Bezeichner steht, kehrt der Kontrollfluss zum Anfang der Schleife zurück. Bei einer solchen unbezeichneten `continue`-Anweisung werden die restlichen Anweisungen im innersten Block der aktuellen Wiederholungsanweisung übersprungen und die Schleife wieder von vorne durchlaufen, beispielsweise um Fehler abzufangen.

Für den Fall einer bezeichneten Stelle in einer äußeren Schleife und Angabe der Bezeichnung bei der `continue`-Anweisung springt der Kontrollfluss an die angegebene Markierung. Dies ermöglicht eine Kontrolle darüber, mit welcher Ebene von verschachtelten Iterationsanweisungen fortgefahren werden soll.

Warnungen

- Eine `continue`-Anweisung darf nur in einem Unteranweisungsblock einer Iterationsanweisung stehen (`while`, `do` oder `for`).
- Auch bei der bezeichneten `continue`-Anweisung gilt, dass ein eventuell vorhandener `finally`-Teil einer umgebenden `try`-Anweisung in der aktiven verschachtelten Ebene immer zuerst ausgeführt wird.

Beispiele

Verhinderung einer Division durch Null mit einer unbezeichneten `continue`-Anweisung:

```
double ergebnis;
for (int teiler = -4; teiler < 4; teiler++)
{
      if (teiler ==0) continue;
      ergebnis= 17.0 / teiler;
      System.out.println(ergebnis);
}
```

Eine bezeichnete `continue`-Anweisung. Aus der innersten Schleife wird zur äußersten Schleife gesprungen:

```
// Verschachtelte Schleifen
weiter:
for (int zahl1 = 1; zahl1 < 3; zahl1++)
{
 for (int zahl2 = 1; zahl2 < 4; zahl2++)
 {

       for (int teiler = -3; teiler < 3; teiler++)
       {
              if (teiler ==0) continue weiter;
              ergebnis= (17.0 * zahl1 * zahl2) / teiler;
              System.out.println(ergebnis);
       } // Ende innere Schleife
 } // Ende zweite Schleife
} // Ende äußere Schleife
```

Verwandte Befehle

`break`
`return`
`throw`

default

Ein optionaler Bestandteil der `switch-case-default`-Auswahlanweisung.

Beschreibung

Über das optionale `default`-Label lässt sich im Rahmen einer `switch-case-default`-Auswahlanweisung ein Block oder eine Unteranweisung angeben, der/die ausgeführt wird, wenn keine der vorher überprüften Werte mit einer explizit angegebenen Bezeichnungskonstante (festgelegt über `case`) übereinstimmt. Der Zweig wird als letzter Bestandteil der `switch-case-default`-Auswahlanweisung notiert.

Anwendung

Sinn macht die `default`-Anweisung nur in Verbindung mit der `switch-case`-Struktur. Diese ermöglicht das Weitergeben des Kontrollflusses an eine von vielen Anweisungen in ihrem Block mit Unteranweisungen. Diese werden dann mit `case` eingeleitet.

An welche `case`-Anweisung innerhalb der `switch`-Anweisung der Kontrollfluss weitergereicht wird, hängt vom Wert des Ausdrucks in der `switch`-Anweisung ab. Es wird die erste Anweisung nach einer `case`-Bezeichnung ausgeführt, welche denselben Wert wie der in der `switch`-Anweisung getestete Ausdruck hat. Wenn es keine entsprechenden Werte gibt, wird die erste Anweisung hinter der `default`-Bezeichnung ausgeführt. Wenn auch die nicht vorhanden ist, wird die erste Anweisung nach dem `switch`-Block ausgeführt.

Beispiele

Angabe eines Defaultzweiges in einer Auswahl:

```
int test; // Eine numerische Variable
...
switch (test)
  {
  case 0: System.out.println(0); break;
  case 1: System.out.println(1);
break;
  case 2: System.out.println(2);
break;
  case 3: System.out.println(3);
break;
  case 4: System.out.println(4);
break;
  case 5: System.out.println(5);
break;
  case 6: System.out.println(6);
break;
```

```
  case 7: System.out.println(7);
break;
  default:
      System.out.println("Kein Treffer");
  }
 }
```

Verwandte Befehle

switch
break
case
if
if-else

do

Eine der drei Iterationsanweisungen von Java.

Beschreibung

Die `do`-Anweisung gibt an, unter welchen Bedingungen eine Schleife gestartet und abgebrochen werden soll.

Anwendung

Die `do`-Anweisung testet eine boolesche Variable oder einen Ausdruck. Solange der Test den Wert `true` liefert, wird die nachfolgende Unteranweisung oder der nachfolgende Block ausgeführt. Erst wenn die boolesche Variable oder der Ausdruck den Wert `false` ausweist, wird die Wiederholung eingestellt und die Schleife verlassen. Es gilt folgende Syntax:

```
do
[Unteranweisung oder Block]
while(Bedingung)
```

Warnungen

- Der Codeblock innerhalb der `do`-Anweisung wird auf jeden Fall mindestens einmal ausgeführt. Dies geschieht immer, ob die Bedingung erfüllt ist oder nicht. Rein von der Syntax her kann man es sich dadurch verdeutlichen, dass die Überprüfung am Ende der Struktur steht. Dies ist ein ganz wichtiger Unterschied zur verwandten `while`-Schleife.
- Die Syntax
  `do [Unteranweisung oder Block] while(true)`
  ist syntaktisch vollkommen korrekt, aber dadurch wird eine Endlosschleife erzeugt. Ebenso entsteht eine Endlosschleife, wenn eine als Testkriterium überprüfte Variable innerhalb der Schleife überhaupt nicht verändert oder so verändert wird, dass das Abbruchkriterium nie erreicht wird.

Beispiele

Eine einfache Schleifenstruktur:

```
do
  {
  ...
  }
while (testvariable == true);
```

Tips

Die Erzeugung einer Endlosschleife (wie oben beschrieben) kann ein Fehler sein, aber auch sinnvoll verwendet werden, wenn im Rahmen von Multithreading darin ein Thread permanent laufen soll.

Verwandte Befehle

`for`
`while`

double

Ein Datentyp (Gleitzahltyp).

Beschreibung

Der Datentyp `double` ist mit einer Länge von 64 Bit der größte Wertebereich von Java. Er dient zur Darstellung von Gleitkommazahlwerten und hat ein Vorzeichen. Der Wertebereich liegt ungefähr zwischen +/- 1,8E+308 nach dem IEEE-754-1985 Standard. Defaultwert ist 0.0. Es existiert wie beim verwandten Typ `float` ein Literal zur Darstellung von *plus/minus Unendlich*, sowie der Wert `NaN` (Not a Number) zur Darstellung von nichtdefinierten Ergebnissen.

Anwendung

Der Datentyp kann wie die meisten Datentypen in Java vielseitig eingesetzt werden. Wie der verwandte Datentyp `float` wird `double` Gleitzahlliteral oder Gleitpunktliteral genannt. Der Dezimalpunkt trennt Vor- und Nachkommateil der Gleitzahl. Standardeinstellung bei Gleitzahlliteralen ist `double`. Wenn ein Gleitzahlliteral als `float` interpretiert werden soll, muss ein `f` oder ein `F` angehängt werden.

Negativen Gleitzahlen wird ein Minuszeichen vorangestellt. Mit dem nachgestellten `e` oder `E`, gefolgt von einem Exponenten (ein negativer Exponent ist ebenso erlaubt), können für Gleitzahlliterale Exponenten verwendet werden. Allerdings nur dann, wenn kein Dezimalpunkt vorhanden ist. Grundsätzlich wird mit diesem Schlüsselwort einer Variablen ein Datentyp oder einer Methode ein Rückgabetyp zugewiesen.

Warnungen

- Bei Operationen mit wenigstens einem Fließkomma-Operanden wird, wenn einer der Operanden den Datentyp `double` hat, der andere ebenso zu `double` konvertiert, und das Ergebnis ist dann ebenfalls vom Typ `double`; sonst werden beide Operanden zum Datentyp `float` konvertiert. Das Ergebnis des Ausdrucks ist dann ebenfalls vom Typ `float`.
- Der Datentyp `double` ist ein gefährlicher Kandidat für Rundungsprobleme (grundsätzlich ein Problem bei Nachkommadarstellungen). Bei arithmetischen Operationen mit Variablen dieses Typs kann es im Nachkommabereich leicht zu Rundungsungenauigkeiten kommen. Das Beispiel bei der Behandlung von `for`-Schleifen mit einer `double`-Zählvariablen zeigt ein solches Problem. Obwohl eine Zählvariable in der Schleife um exakt den Wert `0.05` erhöht wird, wird intern an der letzten Stelle der Darstellung ein Rundungsüberlauf erfolgen, wie in dem Beispiel die Ausgabe beweist.

Beispiele

Eine Variable vom Typ `double` mit einer Zuweisung:

```
double gleitzahl = 0.123;
```

Verwandte Befehle

Die übrigen Datentypen von Java:

```
boolean
byte
char
float
int
long
short
```

else

Das Schlüsselwort bezeichnet den Beginn der Alternative bei einer `if-else`-Auswahlanweisung oder einer `if-else-if`-Anweisung. Das Schlüsselwort kann nur in Zusammenhang mit dem Schlüsselwort `if` verwendet werden.

Beschreibung

Über eine Auswahlanweisung wird einer von mehreren möglichen Kontrollflüssen ausgesucht. Die Anweisungen nach `else` werden ausgeführt, wenn eine Überprüfung in der `if`-Struktur nicht den Wert wahr ergibt.

Anwendung

Sinn macht die `else`-Anweisung nur in Verbindung mit der `if`-Anweisung. Eine `if-else`-Anweisung ist eine Erweiterung der einfachen `if`-Anweisung. Sie besitzt zusätzlich den mit `else` eingeleiteten Block beziehungsweise eine damit eingeleitete Unteranweisung. Dieser `else`-Teil muss nach dem `if`-Zweig notiert werden und wird genau dann ausgeführt, wenn die Überprüfung der Bedingung im `if`-Teil der Anweisung den booleschen Wert `false` liefert. Bei mehr wie zwei Alternativen gibt es einen Spezialfall der `if-else`-Anweisung – die `if-else-if`-Anweisung. Dort folgt innerhalb des `else`-Zweiges einer `if-else`-Anweisung als erstes eine weitere `if`-Anweisung. Es können beliebig viele `else-if`-Anweisungen nacheinander folgen. Für den Fall, wo eine Anweisung immer dann ausgeführt werden soll, wenn keine der vorherigen Überprüfungen den booleschen Wert `true` liefert (ohne explizite Überprüfung auf eine konkrete Bedingung), kann ein optionaler einfacher `else`-Zweig am Ende der Struktur folgen.

Warnungen

- Die einzelnen einer `if`-Anweisung folgenden Blöcke können beliebig kombiniert werden, nur darf nach einem einfachen `else`-Block keine weitere `else`- oder `if-else`-Anweisung mehr folgen.
- Es ist durchaus erlaubt, dass gleiche Werte in einer `if-else-if`-Anweisung mehrfach überprüft werden. Dies ist aber in der Regel nicht sinnvoll, weil bereits beim ersten Treffer der zugehörige Zweig ausgeführt wird und der nachfolgende Zweig (wo die Bedingung ebenso erfüllt wäre) nie erreicht wird.

Beispiele

Eine einfache `if-else`-Struktur:

```
boolean test; // Eine boolesche Variable
...
if (test)
{
...
}
else
{
...
}
```

Eine `if-else-if`-Struktur mit abschließendem einfachen `else`-Zweig. Beachten Sie die syntaktische korrekte (aber unsinnige) doppelte Überprüfung auf `(test==5)`:

```
int test; // Eine numerische Variable
...
 if (test==0)
      {...}
 else if (test==1)
      {...}
 else if (test==3)
      {...}
 else if (test==4)
      {...}
 else if (test==5)
      {...}
 else if (test==6)
      {...}
 else if (test==5)
      {...}
 else
      {...}
```

Tips

Ein Konstrukt mit mehreren `if-else-if`-Alternativen kann für eine etwas größere Auswahl von Möglichkeiten durchaus Sinn machen. Es gibt jedoch eine meist besser geeignete Auswahlanweisung – die `switch`-Anweisung.

Verwandte Befehle

```
switch
break
case
default
if
```

extends

Das Schlüsselwort ist Grundlage für die Angabe der Superklasse einer Klasse bei einer Klassendeklaration beziehungsweise die Angaben einer oder mehrerer Schnittstellenerweiterung(en) bei der Deklaration einer Schnittstelle.

Beschreibung

Java unterstützt als streng objektorientierte Sprache das Prinzip der Vererbung (Einfachvererbung) und die Erweiterung einer Superklasse. Mit dem Schlüsselwort `extends` wird in einer Klassendeklaration die Klasse spezifiziert, auf welcher die neue Klasse aufbaut (die Superklasse).

In Java ist jede Klasse letztendlich eine Ableitung einer einzigen obersten Klasse – der Klasse `java.lang.Object`. Um aber in Java ein konsequent objektorientiertes System einzuhalten, ist ja sogar diese oberste Klasse selbst die Ableitung einer Klasse, einer sogenannten Metaklasse.

Wenn eine Klasse nicht über `extends` explizit als Erweiterung irgendeiner anderen Klasse deklariert ist, wird grundsätzlich angenommen, dass sie eine Erweiterung der Klasse `java.lang.Object` darstellt.

Das Schlüsselwort kommt neben der Angabe der Superklasse einer Klasse auch bei Schnittstellen zum Einsatz. Über eine durch Kommata getrennte Liste von Schnittstellenbezeichnern lassen sich damit Schnittstellenerweiterungen (Superinterfaces) angeben.

Anwendung

Durch die Vererbung einer Superklasse entsteht eine Kopie dieser Klasse, welche dann erweitert werden kann. Generell haben Klassendeklarationen in Java folgendes Format:

```
[modifiers] class NeueKlasse [extends NamederSuperKlasse] [implements
NamederSchnittstelle(n)]
```

wobei alles in eckigen Klammern optional ist.

Bei der Angabe einer oder mehrerer Superschnittstelle(n) im Rahmen einer Schnittstellendeklaration gilt folgende Syntax:

```
[public] interface NamederSchnittstelle [extends SchnittstellenListe]
```

wobei alles in eckigen Klammern optional ist.

Warnungen

In Java gibt es keine Mehrfachvererbung. Deshalb können Java-Klassen immer nur eine Superklasse haben.

Beispiele

Eine Klassendeklaration, welche eine Superklasse erweitert:

```
class MeineKlasse extends meineSuperklasse
```

Ein Applet, welches immer als Erweiterung von `java.applet.Applet` deklariert werden muss:

```
public class MeinApplet extends java.applet.Applet
```

Eine Schnittstellendeklaration mit zwei Superschnittstellen:

```
interface MeineSchnittstelle extends Superschnittstelle1, Superschnittstelle2
```

Verwandte Befehle

`implements`

false

Ein boolesches Literal.

Beschreibung

Das Schlüsselwort `false` beschreibt eine von zwei möglichen Ausprägungen von booleschen Variablen oder Ausdrücken. Es steht für falsch.

Anwendung

Java kennt nur zwei boolesche Literale: `true` und `false`. Diese sind die einzigen möglichen Ausprägungen von booleschen Variablen oder Ausdrücken.

Warnungen

Im Gegensatz zu vielen anderen Sprachen gibt es keine direkte Äquivalenz zwischen numerischen Werten ungleich 0 beziehungsweise gleich 0 und den beiden booleschen Literalen.

Beispiele

Eine boolesche Variablendeklaration:

```
boolean test=false;
```

Verwandte Befehle

```
boolean
true
```

final

Ein Modifier für Klassen, Variablen und Methoden.

Beschreibung

Dieses Schlüsselwort dient als Modifier, um Klassen, Variablen und Methoden als `final` zu kennzeichnen. Er wird der Deklaration einfach vorangestellt (am Beginn der Deklaration).

Anwendung

Der Modifier `final` legt einen endgültigen Standard fest, welcher nicht mehr verändert werden soll. Der Modifier kann sowohl auf Klassen als auch Methoden und Variablen angewendet werden. Dabei ist die Wirkung für die drei Fälle zu unterscheiden:

- Bei dem Klassenmodifier `final` gilt folgendes: Finale Klassen können nicht weiter abgeleitet werden, das bedeutet, sie können keine Subklassen haben.
- Das Schlüsselwort `final` vor einer Methodendeklaration verhindert, dass irgendwelche Subklassen der diese Methode implementierenden Klasse die so gekennzeichnete Methode überschreiben.
- Am komplexesten und umfangreichsten ist der Fall, wo `final` auf Variablen angewendet werden soll. In Java werden Konstanten (Speicherbereiche mit Werten, die sich nie ändern) ausschließlich(!) über das Voranstellen des Schlüsselwortes `final` bei einer Variablen realisiert. Zusätzlich muss dieser damit als nicht mehr veränderbar deklarierten Variablen ein Anfangswert zugewiesen werden. Dies geht sowohl für Instanzvariablen als auch Klassenvariablen, aber nie für lokale Variablen (Variablen, welche innerhalb von Methodendefinitionen deklariert und auch nur dort benutzt werden).

Warnungen

- Bei einer Verwendung von `final` als Klassenmodifier gilt, dass eine Klasse nicht sowohl `final` als auch `abstract` als Modifier haben kann. Bei der Verwendung in Zusammenhang mit Methoden gilt analog, dass abstrakte Methoden nie als final deklariert werden, weil dadurch verhindert wird, dass diese Methoden überschrieben werden können. Sie wären nutzlos.
- Der Modifier `final` darf bei der Deklaration einer Methode in einer Schnittstelle nicht verwendet werden. Hingegen gilt für Variablen in Schnittstellen genau das Gegenteil. Alle Variablen, die in einer Schnittstelle deklariert werden, sind immer `public`, `final` und `static`. Das muss nicht explizit in der Variablendeklaration angegeben werden, obwohl es der besseren Lesbarkeit halber sinnvoll ist. Weil aber damit alle Variablen in Schnittstellen `final` sind, müssen sie

bereits in der Schnittstelle unbedingt initialisiert werden. Achtung – eine automatische Initialisierung über den Defaultwert findet nicht statt. Es wäre auch sinnlos.

- Lokale Variablen dürfen nie `final` deklariert werden.

Beispiele

Eine finale Klasse:

```
final class test
  {... }
```

Eine finale Methode:

```
final void meineMethode();
```

Eine finale Variable mit direkter Wertzuweisung (das bedeutet eine Konstante):

```
final int antwort = 42;
```

Verwandte Befehle

```
abstract
public
```

finally

Ein optionaler Teil der Schutzanweisung zum Auffangen von Ausnahmen in Java.

Beschreibung

Java verfolgt zum Abfangen von Laufzeitfehlern ein Konzept, welches mit sogenannten Ausnahmen arbeitet. Ausnahmen werden unter Java über ein `try-catch-finally`-Konstrukt aufgefangen. Die beiden Schlüsselworte `try` und `catch` müssen dabei immer auftreten, `finally` ist optional. Dabei wird der mit `finally` eingeleitete Zweig zum Definieren der Anweisungen verwendet, die immer – unabhängig vom Auftreten einer Ausnahme – am Ende eines Vorgangs abgearbeitet werden sollen. Das Schlüsselwort kann nur in Zusammenhang mit einem vorangestellten `try-catch`-Konstrukt verwendet werden.

Anwendung

Während der Laufzeit eines Programm kann es nichtplanbare Situationen geben, welche zur Laufzeit eines Programms auftreten und eine unmittelbare Reaktion durch das Programm beziehungsweise den Anwender notwendig machen. Etwa, wenn ein Zugriff auf ein Diskettenlaufwerk erfolgt, wo keine Diskette eingelegt ist, oder die Division durch Null während einer mathematischen Operation. Ein solcher Vorgang wird in Java eine Ausnahme erzeugen, welche dann unmittelbar vom Programm bearbeitet werden muss. Damit diese Ausnahmen sicher gehandhabt werden können, stellt Java Schutzanweisungen zur Verfügung. Diese Anweisungen benutzen im Wesentlichen die drei Schlüsselworte `try`, `catch` und `finally`. Innerhalb des mit `try` eingeleiteten Blocks werden die Anweisungen notiert, welche eine Ausnahme erzeugen können. Kritische Aktionen in einem Java-Programm sollten immer innerhalb eines `try`-Blocks durchgeführt werden.

Innerhalb des oder der mit `catch` eingeleiteten Blocks/Blöcke werden die Anweisungen notiert, welche beim Auftreten einer bestimmten Ausnahme durchgeführt werden sollen.

Die optionale `finally`-Anweisung erlaubt die Abwicklung wichtiger Abläufe (wie etwa das Schließen von Dateien) bevor die Ausführung des gesamten `try-catch-finally`-Konstruktes beendet wird. Der in dem `finally`-Block untergebrachte Code wird in jedem Fall abgearbeitet (unabhängig von dem Auftreten einer Ausnahme).

Beispiele

Eine Ausnahmebehandlungsstruktur mit `finally`-Teil:

```
try
{
//  kritische Anwendungen
}
catch (Exception e)
{
// Behandlung der Ausnahme
// Das Ausnahmeobjekt e wird behandelt
}
finally
{
// Abschlußarbeiten
}
```

Verwandte Befehle

```
try
catch
switch
break
default
if
if-else
```

float

Ein Datentyp (Gleitzahltyp).

Beschreibung

Der Datentyp `float` ist mit einer Länge von 32 Bit der kleinste Wertebereich von Java zur Darstellung von Gleitkommazahlwerten. Er hat ein Vorzeichen. Der Wertebereich liegt ungefähr zwischen +/- 3,4E+38 nach dem IEEE-754-1985-Standard. Defaultwert ist 0.0. Es existiert wie beim verwandten Typ `double` ein Literal zur Darstellung von *plus/minus Unendlich*, sowie der Wert `NaN` (Not a Number) zur Darstellung von nicht definierten Ergebnissen.

Anwendung

Java verwendet bei Gleitzahlliteralen oder Gleitpunktliteralen (die beiden Gleitzahltypen `float` und `double`) als Standardeinstellung `double`. Der Dezimalpunkt trennt Vor- und Nachkommateil der Gleitzahl. Wenn ein Gleitzahlliteral als `float` interpretiert werden soll, muss ein `f` oder ein `F` angehängt werden. Negativen Gleitzahlen wird ein Minuszeichen vorangestellt. Mit dem nachgestellten `e` oder `E`, gefolgt von einem Exponenten (ein negativer Exponent ist ebenso erlaubt), können für Gleitzahlliterale Exponenten verwendet werden. Allerdings nur dann, wenn kein Dezimalpunkt vorhanden ist.

Der Datentyp kann in Java so vielseitig eingesetzt werden, dass eine genauere Beschreibung der potentiellen Anwendungen den Rahmen sprengt. Grundsätzlich wird mit diesem Schlüsselwort einer Variablen ein Datentyp oder einer Methode ein Rückgabetyp zugewiesen.

Warnungen

Bei Operationen mit wenigstens einem Fließkomma-Operanden wird, wenn einer der Operanden den Datentyp `double` hat, der andere ebenso zu `double` konvertiert, und das Ergebnis ist dann ebenfalls vom Typ `double`; sonst werden beide Operanden zum Datentyp `float` konvertiert. Das Ergebnis des Ausdrucks ist dann ebenfalls vom Typ `float`.

Beispiele

Eine Variable vom Typ `float` mit einer Zuweisung:

```
float gleitzahl = 0.123;
```

Verwandte Befehle

Die übrigen Datentypen von Java:

```
boolean
byte
char
double
int
long
short
```

for

Eine der drei Iterationsanweisungen von Java.

Beschreibung

Die `for`-Anweisung gibt an, wie oft eine Schleife durchlaufen werden soll.

Anwendung

Die `for`-Anweisung testet eine boolesche Variable oder einen Ausdruck. Solange der Test den Wert `true` liefert, wird die nachfolgende Unteranweisung oder der nachfolgende Block ausgeführt. Erst wenn die boolesche Variable oder der Ausdruck den Wert `false` ausweist, wird die Wiederholung eingestellt und die Schleife verlassen.

Es gilt folgende Syntax:

```
for ([Initialisierungsteil]; [Testteil];[In- oder Dekrementteil])
[Unteranweisung oder Block]
```

Bei der Syntax gelten folgende Regeln:

- Hinter `for` kann optional ein Leerzeichen folgen oder auch direkt die nachfolgend zu notierende Klammer.
- Der klassische (aber nicht zwingende) Aufbau einer `for`-Schleife sieht folgendermaßen aus: im Initialisierungsteil des Schleifenkopfes wird einer Zählvariablen ein Anfangswert zugewiesen, der dann im Testteil überprüft und im dritten Teil des Schleifenkopfes verändert wird. Der Initialisierungswert sollte kleiner (oder gleich) dem Wert sein, welcher im Testteil als Abbruchkriterium getestet wird, sofern in der Schleife inkrementiert (hochgezählt) wird. Sofern jedoch dekrementiert (runtergezählt) wird, gilt genau die Umkehrung.
- Als Zählvariable ist außer `short` jeder numerische Datentyp erlaubt (auch Gleitzahltypen). Defaulteinstellung ist `int`.
- Es sind mehrere Tests in einer `for`-Schleife möglich. Der Initialisierungsteil kann eine durch Kommata getrennte Reihe von Deklarations- und Zuweisungsanweisungen enthalten. Erst durch ein Semikolon wird der Initialisierungsteil beendet. Diese Deklarationen haben nur Gültigkeit für den Bereich der `for`-Anweisung und ihrer Unteranweisungen. Auch der In- oder Dekrementteil kann eine durch Kommata getrennte Reihe von Ausdrücken sein, die einmal pro Durchlauf der Schleife bewertet werden. Diese mehrfachen Ausdrücke in diesem Teil der `for`-Schleife machen nur dann Sinn, wenn sie bereits im Initialisierungsteil deklariert wurden. Dieser Teil wird für gewöhnlich dazu verwendet, einen Index, der im Testteil überprüft wird, zu inkrementieren (hochzuzählen) oder zu dekrementieren (herabzuzählen). Auch hier steht am Ende ein Semikolon. Beachten Sie bitte, dass – sofern mehrere Testvariablen überprüft werden – diese gemeinsam pro Durchlauf bewertet werden. Es ist auch nicht möglich, im Testteil durch Komma getrennte Tests durchzuführen. Der Testteil enthält also genau eine boole-

sche Variable oder einen Ausdruck, der einmal pro Schleifendurchlauf neu bewertet wird. Wenn der Vergleich den Wert `false` ergibt, wird die `for`-Schleife verlassen. Auch dieser Teil wird wieder durch ein Semikolon beendet.

- Die Zählvariablen in `for`-Schleifen müssen nicht unbedingt außerhalb der Schleife vereinbart werden, sondern können ebenso im Initialisierungsteil der `for`-Schleife direkt deklariert werden und sind dann auch nur dort bekannt. Diese im Initialisierungsteil der `for`-Schleife direkt vereinbarten Zählvariablen heißen schleifenlokale Variablen und sind ein Sonderfall von normalen lokalen Variablen. Sie überdecken ggf. gleichnamige Variablen im übergeordneten Programmblock. Die Variablen im übergeordneten Programmblock bleiben hierdurch unverändert.

Warnungen

- Die Syntax `for(;;)`, das bedeutet ohne explizite Parameter, wird keinen Compilerfehler erzeugen, denn syntaktisch ist sie völlig korrekt. Es wird eine Endlosschleife erzeugt.
- Ein falsches De- beziehungsweise Inkrementieren zählt zu den häufigsten Fehlerquellen in der `for`-Schleife. Der Compiler wird dies nicht überprüfen und – sofern keine Syntaxverletzung vorliegt – keine Fehlermeldung ausgeben. Auch ein Überprüfen auf eine falsche Bedingung im Testteil veranlasst den Compiler (bei korrekter Syntax) zu keiner Meldung.
- Java unterstützt, außer in Initialisierungs- und Weiterführungsklauseln für Schleifen-Anweisungen, keine mit Kommata zusammengesetzten Anweisungen.

Beispiele

Eine einfache `for`-Schleife, die eine Zählvariable inkrementiert:

```
int i;
for (i=0; i < 42;i++)
{
// tue was
}
```

Eine einfache `for`-Schleife, die eine Zählvariable dekrementiert:

```
int i;
for (i=255; i >= 0;i--)
{
// tue was
}
```

Eine `for`-Schleife mit schleifenlokaler Zählvariable:

```
for (int i=0; i < 255;i++)
{
// tue was
}
```

Zwei verschachtelte Schleifen:

```
int i, j;
for (i=0;i < 3; i++)
{
// tue was in der ersten Ebene
    for (j=0;j < 3; j++)
    {
      // tue was in der inneren Schleife
    }
}
```

Zwei Testvariablen – sie werden gemeinsam pro Durchlauf bewertet, wie die zwei aufeinanderfolgenden Schleifen deutlich machen:

```
int j,k;
for (k=1, j=1; j+k < 5; j++, k++)
{
System.out.println("j = " + j);
System.out.println("k = " + k);
System.out.println("k + j  = " + (k + j));
}
for (k=1, j=1; j < 5; j++, k=k + 2)
{
System.out.println("j = " + j);
System.out.println("k = " + k);
System.out.println("k + j  = " + (k + j));
}
```

Eine Schleife mit `double`-Zählvariablen:

```
double j,k;
for (k=1, j=1; j+k < 5; j++, k=k+0.5)
{
// tue was
}
```

Tips

Die Erzeugung einer Endlosschleife per `for`-Anweisung kann ein Fehler sein, aber auch sinnvoll verwendet werden, wenn im Rahmen von Multithreading darin ein Thread permanent laufen soll. Besser als eine per `for` erzeugte Endlosschleife eignet sich aber eine mit `while(true)` oder `do-while(true)` erzeugte Endlosschleife.

Verwandte Befehle

`do`
`while`

goto

Ein in der aktuellen Java-Version reserviertes Schlüsselwort, das derzeit ohne Funktionalität ist.

Beschreibung

Es gibt in Java einige Token, die vorsorglich für spätere Versionen von Java reserviert wurden, sonst aber derzeit noch keine weitere Bedeutung haben. Es handelt sich um die Token `byvalue`, `cast`, `const`, `future`, `generic`, `goto`, `inner`, `operator`, `outer`, `rest` und `var`. Von diesen Token werden aber nur `const` und `goto` offiziell zu den Schlüsselworten von Java gezählt.

Anwendung

Derzeit ohne konkrete Anwendung

Verwandte Befehle

Die übrigen reservierten Token ohne konkrete Funktionalität (verwandt im Sinne von »derzeit ohne konkrete Funktionalität«).

if

Eine Auswahlanweisung.

Beschreibung

Über eine `if`-Auswahlanweisung wird entschieden, ob ein nachfolgender Block beziehungsweise eine Unteranweisung ausgeführt werden soll oder nicht. In Verbindung mit dem optionalen Schlüsselwort `else` oder einer beziehungsweise mehreren `if-else-if`-Anweisung(en) kann einer von mehreren möglichen Kontrollflüssen ausgesucht werden.

Anwendung

Eine `if`-Auswahlanweisung besitzt eine Kontrollstruktur, in welcher eine boolesche Variable oder der Rückgabewert eines Vergleichs überprüft werden. Der nachfolgende Block beziehungsweise eine Unteranweisung wird genau dann ausgeführt, wenn die Überprüfung der Bedingung im `if`-Teil der Anweisung den booleschen Wert `true` liefert. Andernfalls wird direkt nach dem Block beziehungsweise der Unteranweisung mit dem Programm fortgefahren. Dort lassen sich dann mit dem optionalen Schlüsselwort `else` oder einer beziehungsweise mehreren `if-else-if`-Anweisung(en) alternativ auszuführende Anweisungen notieren.

Warnungen

»Klassischer« Syntaxfehler bei einem Test innerhalb der `if`-Auswahlanweisung ist eine Zuweisung (über =) statt eines Vergleichs (über ==). Im Gegensatz zu C/C++ wird dieser Fehler vom Java-Compiler aber gemeldet.

Beispiele

Eine einfache `if`-Struktur mit einer booleschen Variablen:

```
boolean test; // Eine boolesche Variable
...
if (test)
{
...
}
```

Eine einfache `if`-Struktur mit einer numerischen Variablen:

```
int test; // Eine numerische Variable
...
 if (test==0)
{
...
}
```

Verwandte Befehle

```
switch
break
case
default
else
```

implements

Die optionale Angabe einer oder mehrerer Schnittstellen einer Klasse bei einer Klassendeklaration.

Beschreibung

Java unterstützt als streng objektorientierte Sprache zwar das Prinzip der Vererbung und die Erweiterung einer Superklasse, aber keine Mehrfachvererbung. Statt dessen stellt Java das Prinzip der Schnittstellen zur Verfügung. Mit dem Schlüsselwort `implements` wird in einer Klassendeklaration spezifiziert, welche Schnittstellen implementiert werden.

Anwendung

Generell haben Klassendeklarationen in Java folgendes Format:

```
[modifiers] class NeueKlasse [extends NamederSuperKlasse] [implements
NamederSchnittstelle(n)]
```

wobei alles in eckigen Klammern optional ist.

Eine Schnittstelle ist in der Java-Sprache eine Sammlung von Methodennamen ohne konkrete Definition beziehungsweise die Deklaration von Konstanten. Obwohl eine einzelne Java-Klasse nur genau eine Superklasse haben kann, können in einer Klasse mehrere Schnittstellen implementiert werden. Diese werden über `implements` in der Klassendeklaration angegeben und in der Klasse (beziehungsweise einer deren Subklassen) überschrieben. Eine Schnittstellenmethode kann nicht verwendet werden, bis sie nicht in der Klasse überschrieben worden ist, welche diese Methode implementiert.

Warnungen

- Wenn eine Klasse eine Schnittstelle implementiert, müssen alle in der Schnittstelle deklarierten Methoden in der Klasse überschrieben werden. Falls dies nicht erfolgt, muss die Klasse als abstrakt deklariert werden (der Compiler meldet sonst einen Fehler).
- Alle in Schnittstellen deklarierten Konstanten müssen dort einen Wert zugewiesen bekommen.
- Eine Schnittstelle kann selbst keine Schnittstellen über `implements` implementieren. Dafür gibt es die Erweiterung einer Schnittstelle durch eine oder mehrere Superschnittstelle(n) über das Schlüsselwort `extends`.

Beispiele

Eine Klassendeklaration, welche zwei Schnittstellen implementiert:

```
class MeineKlasse implements MeineSchnittstelle, MeineZweiteSchnittstelle
```

Verwandte Befehle

`extends`

import

Eine Vereinfachung des Zugriffs auf Java-Klassen, -Schnittstellen und -Pakete über einen verkürzten Namen.

Beschreibung

Java verfolgt ein Paket-Konzept, welches dazu dient, mehrere Klassen und/oder Schnittstellen zu einem Paket über die Anweisung `package` zusammenzufassen. Durch die Anweisung `import` werden einzelne Pakete oder konkrete Klassen/Schnittstellen in einem Programm über einen verkürzten Namen verfügbar gemacht.

Anwendung

Pakete (engl. Packages) sind in Java eine Art Analogon zu Bibliotheken vieler anderer Computersprachen. So wird beispielsweise die Java-Laufzeitbibliothek, das Java-API, selbst in Form von Paketen zur Verfügung gestellt. Aber auch jeder Java-Entwickler kann seine eigenen Pakete zusammenstellen (über das Schlüsselwort `package`). Ein Paket von Java-Klassen enthält normalerweise logisch zusammenhängende Klassen und Schnittstellen.

Wenn in einer Java-Klasse nun auf Bestandteile einer anderen Klasse zugegriffen werden soll, kann dies explizit mittels der Punktnotation geschehen (als vollqualifizierte Angabe). Beispielsweise kann eine Klasse durch Angaben sämtlicher Pakete referenziert werden, innerhalb derer sie sich befindet.

Beispiel: `java.applet.Applet`

Wenn nun aus der angegebenen Klasse eine Methode benötigt wird, muss diese Pfadangabe um den Methodennamen erweitert werden. Dies gilt übertragen für jeden anderen Sprachbestandteil von Java. Und zwar jedes Mal, wenn der Zugriff auf einen Bestandteil der fremden Klasse notwendig wird.

Um die Lesbarkeit des Quelltextes und vor allem den Schreibaufwand zu reduzieren, kann man in einem Java-Quelltext häufig benötigte Pakete beziehungsweise einzelne Klassen am Anfang des Quelltextes mit der `import`-Anweisung importieren. Dabei können mehrere `import`-Anweisungen nacheinander notiert werden.

Wenn eine spezielle Klasse oder Schnittstelle explizit referenziert werden soll, gibt man sie mit voll qualifiziertem Namen an. Wenn ein Paket referenziert werden soll, arbeitet man mit Platzhaltern anstelle der Angabe eines expliziten Paketes. Der Platzhalter selbst ist die auch sonst oft übliche Angabe eines Sterns (`*`).

Den `import`-Befehl gibt es in 3 (sinnvollen) Varianten:

```
import paket.klasse;
import paket.*;
import paket;
```

- Die erste Form erlaubt nachfolgend den gezielten Zugriff auf eine in dem referenzierten Paket enthaltene Klasse ohne die per Punktnotation vorangestellte Angabe des Paketes.
- Bei Fall zwei kann nachfolgend eine Klasse innerhalb des Source-Codes ebenfalls direkt über ihren Namen (ohne Angabe der Pakete) angesprochen werden. Der Vorteil gegenüber der ersten Variante ist das Einbinden aller Klassen und Schnittstellen aus einem Paket mit einer Anweisung.
- Die dritte Form erlaubt nachfolgend noch keine direkte Ansprache einer Klasse/Schnittstelle. Man kann nur auf den Teil der Angabe verzichten, der bereits referenziert wird (ein Teil der vollständigen Paketangaben). Bei nachfolgendem Zugriff auf Paketbestandteile muss vor dem Klassen-/Schnittstellennamen das letzte Element aus dem importierten Packagenamen per Punktnotation gesetzt werden. Es handelt sich also strenggenommen um das Bereitstellen eines verkürzten Paketenamens.

Warnungen

- Das Importieren von Paketen/Klassen/Schnittstellen muss vor der Definition irgendeiner Klasse in der Datei stehen. Wenn Sie in einem Paket selbst eine andere Klasse/Schnittstelle importieren wollen, muss die `import`-Anweisung nach der `package`-Anweisung stehen.
- Die `import`-Anweisung dient nur dazu, Java-Elemente über einen verkürzten Namen innerhalb der aktuellen Bezugsklasse zugänglich zu machen und damit den Code zu vereinfachen. Sie hat nicht den Sinn (wie beispielsweise die `include`-Anweisung in C), die Klassen zugänglich zu machen oder sie einzulesen. Insbesondere wird der lauffähige Code in keiner Weise vergrößert oder sonst irgendwie verschlechtert (Performance etc.), wenn überflüssige `import`-Anweisungen verwendet werden. Der Compiler optimiert den Code.
- Beim Importieren von ganzen Paketen mittels des Platzhalters Stern (`*`) werden keine (!) untergeordneten Pakete importiert. Um untergeordnete Pakete zu importieren, müssen diese explizit angegeben werden.

Beispiele

Importieren einer einzelnen Java-Klasse:

```
import java.applet.Applet;
```

Importieren eines Paketes:

```
import java.applet.*;
```

Bereitstellen eines verkürzten Paketnamens:

```
import java.applet;
```

Verwandte Befehle

`package`

instanceof

Ein Typvergleichsoperator.

Beschreibung

Ein binärer Operator, der zwei Operanden daraufhin vergleicht, ob es sich bei dem linken Operanden um eine Instanz der als rechter Operand angegebenen Klasse handelt.

Anwendung

Sofern die Operanden des Operators tatsächlich in einer Klassen-Instanz-Beziehung zueinander stehen, liefert der Operator den Wert `true` zurück, andernfalls `false`.

Warnungen

Das spezifizierte Objekt muss eine unmittelbare Instanz der getesteten Klasse sein, damit der Wert `true` zurückgegeben wird. Der Vergleich zwischen der Instanz und einer Subklasse der erzeugenden Klasse liefert `false`.

Beispiele

Test, von welcher Klasse eine Instanz abstammt:

```
class MeineKlasse
{
...
 }
class MeineKlasseZwei
{
...
MeineKlasse meinObjekt = new MeineKlasse();
...
if (meinObjekt instanceof MeineKlasse)
{
 // tue was
}
}
```

int

Ein Datentyp (Ganzzahltyp).

Beschreibung

Der Datentyp `int` ist der Standardwertebereich von Java zur Darstellung von Ganzzahlwerten (ganzzahliges Zweierkomplement mit Vorzeichen). Er hat eine Länge von 32 Bit. Defaultwert ist 0. Dieser Datentyp kann die Ganzzahlwerte von (– 2 hoch 31 =) -2.147.483.648 bis (+ 2 hoch 31 – 1 =) 2.147.483.647 darstellen.

Anwendung

Diesen Datentyp wird man in Java am häufigsten finden. Prinzipiell wird mit diesem Schlüsselwort einer Variablen ein Datentyp oder einer Methode ein Rückgabetyp zugewiesen. Eine genauere Beschreibung der potentiellen Anwendungen sprengt wie bei allen Datentypen den Rahmen. Die Datentypen `int` und `long` können dezimal, aber auch hexadezimal sowie oktal beschrieben werden. Die Voreinstellung ist dezimal und gilt immer dann, wenn die Werte ohne weitere Angaben dargestellt werden. Hexadezimale Darstellung beginnt immer mit der Sequenz `0x` oder `0X`. Oktale Darstellungen beginnen mit einer führenden Null. Voreinstellung ist immer der Typ `int`. Durch Anhängen von `l` oder `L` kann man jedoch explizit den Typ `long` wählen. Sofern man für einen `int`-Datentyp einen Wert wählt, welcher den zulässigen Wertebereich überschreitet, muss dies sogar erfolgen.

Warnungen

Wenn eine binäre Operation mit zwei Ganzzahl-Operanden durchgeführt wird, ist das Ergebnis immer eine Variable vom Datentyp `int` oder `long`. Das Ergebnis wird nur dann als `long` dargestellt, wenn einer der Operanden vom Datentyp `long` war oder das Ergebnis nicht ohne Überlauf in einer Variablen vom Datentyp `int` darzustellen ist. Die Datentypen `byte` oder `short` können als Ergebnis nur vorkommen, wenn es explizit festgelegt wird.

Beispiele

Eine Methodendeklaration mit einem Rückgabewert vom Typ `int`:

```
int test();
```

Eine Variable vom Typ `int`:

```
int test;
```

Verwandte Befehle

Die übrigen Datentypen von Java:

```
boolean
byte
char
double
float
long
short
```

interface

Der Beginn einer Schnittstellendeklaration.

Beschreibung

Mit diesem Schlüsselwort wird festgelegt, dass es sich um die Deklaration einer Schnittstelle handelt. Generell haben Schnittstellendeklarationen in Java folgendes Format:

```
[public] interface NamederSchnittstelle [extends SchnittstellenListe]
```

wobei alles in eckigen Klammern optional ist. Dabei können auch mehr als nur eine Schnittstelle implementiert werden. Dann müssen mehrere Schnittstellen mit der `extends`-Anweisung hintereinander geschrieben werden (durch Kommata getrennt).

Anwendung

Eine Schnittstelle ist in der Java-Sprache eine Sammlung von Methodennamen ohne konkrete Definition beziehungsweise die Deklaration von Konstanten. Obwohl eine einzelne Java-Klasse nur genau eine Superklasse haben kann, können in einer Klasse mehrere Schnittstellen implementiert werden. Diese werden über `implements` in der Klassendeklaration angegeben und in der Klasse überschrieben. Eine davon abgeleitete Subklasse kann diese Implementierung durch eine eigene Variante überschreiben.

Eine Klasse, die eine Schnittstelle implementiert, muss alle Methoden der Schnittstelle überschreiben. Es ist also nicht möglich, nur einige wenige Methoden »herauszupicken« und die anderen Methoden, welche in der Schnittstelle deklariert werden, zu ignorieren. Dies gilt auch, wenn sie gar nicht benötigt werden (bei umfangreichen Schnittstellen kann das etwas lästig sein; dies ist der Grund für die Bereitstellung von Adapterklassen im awt-Package, wo solche Fälle häufig auftreten können).

Die Syntax zur Erstellung einer Schnittstelle ähnelt der zur Erstellung einer Klasse mit dem wichtigen Unterschied, dass keine Methode in einer Schnittstelle einen Körper haben darf und keine Variablen deklariert werden dürfen, die nicht als Konstanten dienen.

Schnittstellen können als Voreinstellung von allen Klassen im selben Paket implementiert werden (freundliche Einstellung). Damit verhalten sie sich wie freundliche Klassen und Methoden. Der optionale Modifier `public` ermöglicht es den Klassen und Objekten außerhalb des aktuellen Paketes, diese Schnittstelle zu implementieren. Andere Zugriffsmodifier sind nicht erlaubt!

Im Körper einer Schnittstelle werden nur Konstanten deklariert sowie die gewünschten Methoden festgelegt. Eine Methode wird durch Angabe des Namens,

der Parameterliste und ihres Rückgabewerts festgelegt. Da in einer Schnittstelle kein Code definiert wird, gibt es keinen Methodenkörper; die Deklaration der Methode endet mit einem Semikolon.

Variablen in Schnittstellen werden wie finale statische Variablen in Klassen dazu verwendet, Konstanten zu definieren, die allen Klassen zur Verfügung stehen, welche diese Schnittstellen implementieren.

Warnungen

- Wenn eine Schnittstelle von einer Klasse implementiert wird, müssen sämtliche in der Schnittstelle deklarierten Methoden in der Klasse überschrieben werden. Falls dies nicht erfolgt, muss die Klasse als abstrakt deklariert werden (der Compiler meldet sonst einen Fehler).
- Alle in Schnittstellen deklarierten Konstanten müssen dort einen Wert zugewiesen bekommen.
- Wenn Sie eine bereits über eine Superschnittstelle erweiterte Schnittstelle implementieren, müssen sowohl die Methoden der neuen als auch die der Superschnittstelle überschrieben werden.
- Analog zu `public`-Klassen müssen `public`-Schnittstellen zwingend in einer Datei namens `<NamederSchnittstelle>.java` definiert werden.
- Eine Schnittstelle kann selbst keine Schnittstellen über `implements` implementieren. Sie kann jedoch Methoden und Konstanten von anderen Schnittstellen erben über das Schlüsselwort `extends`.
- Der Modifier `final` darf bei der Deklaration einer Methode in einer Schnittstelle nicht verwendet werden. Hingegen gilt für Variablen in Schnittstellen genau das Gegenteil. Alle Variablen, die in einer Schnittstelle deklariert werden, sind immer `public`, `final` und `static`. Das muss nicht explizit in der Variablendeklaration angegeben werden, obwohl es der besseren Lesbarkeit halber sinnvoll ist. Weil aber damit alle Variablen in Schnittstellen final sind, müssen sie bereits in der Schnittstelle unbedingt initialisiert werden. Achtung – eine automatische Initialisierung über den Defaultwert findet nicht statt. Es wäre auch sinnlos.

Beispiele

Eine minimale Schnittstellendeklaration:

```
interface MeineSchnittstelle
{... }
```

Eine Schnittstellendeklaration, welche eine Superschnittstelle erweitert:

```
interface MeineSchnittstelle extends meineSuperschnittstelle
{...}
```

Tips

Im Falle von Schnittstellen ist der Modifier `abstract` bei der Deklaration von Methoden nicht explizit notwendig, denn alle Schnittstellen sind abstrakt. Der Modifier kann jedoch trotzdem zur Eindeutigkeit gesetzt werden.

Verwandte Befehle

`extends`
`class`
`implements`

long

Ein Datentyp (Ganzzahltyp).

Beschreibung

Der Datentyp `long` ist der größte Wertebereich von Java zur Darstellung von Ganzzahlwerten (ganzzahliges Zweierkomplement mit Vorzeichen). Er hat eine Länge von 64 Bit. Defaultwert einer Variablenn vom Typ `long` ist 0. Dieser Datentyp kann die Ganzzahlwerte von -9.223.372.036.854.775.808 (- 2 hoch 63) bis 9.223.372.036.854.775.807 (+ 2 hoch 63 – 1) darstellen.

Anwendung

Die Datentypen `int` und `long` können dezimal, aber auch hexadezimal sowie oktal beschrieben werden. Die Voreinstellung ist dezimal und gilt immer dann, wenn die Werte ohne weitere Angaben dargestellt werden. Hexadezimale Darstellungen beginnen immer mit der Sequenz `0x` oder `0X`. Oktale Darstellungen beginnen mit einer führenden Null. Voreinstellung ist immer der Typ `int`. Durch Anhängen von `l` oder `L` kann man jedoch explizit den Typ `long` wählen. Sofern man für einen `int`-Datentyp einen Wert wählt, welcher den zulässigen Wertebereich überschreitet, muss dies sogar erfolgen. Der Datentyp kann in Java so vielseitig eingesetzt werden, dass eine genauere Beschreibung der potentiellen Anwendungen den Rahmen sprengt. Prinzipiell wird mit diesem Schlüsselwort einer Variablen ein Datentyp oder einer Methode ein Rückgabetyp zugewiesen.

Warnungen

Wenn eine binäre Operation mit zwei Ganzzahloperanden durchgeführt wird, ist das Ergebnis immer eine Variable vom Datentyp `int` oder `long`. Das Ergebnis wird nur dann als `long` dargestellt, wenn einer der Operanden vom Datentyp `long` war oder das Ergebnis nicht ohne Überlauf in einer Variablen vom Datentyp `int` darzustellen ist. Die Datentypen `byte` oder `short` können als Ergebnis nur vorkommen, wenn es explizit festgelegt wird.

Beispiele

Eine Methodendeklaration mit einem Rückgabewert vom Typ `long`:

```
long test();
```

Eine Variable vom Typ `long`:

```
long test;
```

Verwandte Befehle

Die übrigen Datentypen von Java:

```
boolean
byte
char
double
float
int
short
```

native

Ein Methodenmodifier.

Beschreibung

Über den Modifier `native` werden Methoden deklariert, welche nicht in Java geschrieben sind, aber dennoch innerhalb von Java-Code verwendet werden sollen.

Anwendung

Native Methoden sind Methoden, die nicht in Java geschrieben sind, aber dennoch innerhalb von Java verwendet werden sollen. Meist handelt es sich um Methoden in C/C++. Die Syntax zum Verwenden von nativen Methoden ist ähnlich zu abstrakten Methoden. Der Modifier `native` wird vor der Methode deklariert, und der Body (Körper) der Methode wird durch ein Semikolon ersetzt.

Es muss bei der Verwendung von `native` beachtet werden, dass die Deklaration den Compiler über die Eigenschaften der Methode informiert. Die Deklaration der Methode muss daher hinsichtlich Rückgabewert und Parameterliste mit der nativen Methode übereinstimmen.

Beispiele

Eine native Methodendeklaration:

```
native void meineNativeMethode();
```

Verwandte Befehle

`abstract`

new

Die Erstellung eines neuen Objektes.

Beschreibung

Über das Schlüsselwort `new` werden in Java neue Instanzen einer Klasse erstellt.

Anwendung

Das Schlüsselwort `new` ist eng verbunden mit dem Begriff der Konstruktoren (engl. Constructors) beziehungsweise Konstruktormethoden. Konstruktoren sind spezielle Methoden einer Klasse, die zur Erstellung einer Klasseninstanz dienen. Der über das Schlüsselwort spezifizierte `new`-Operator wird zusammen mit einem Klassennamen aufgerufen. Dadurch wird implizit ein geeigneter Konstruktor aufgerufen und die gewünschte Klasseninstanz angelegt und initialisiert. Prinzipiell werden sämtliche für die Erstellung eines Objektes notwendigen Schritte vollzogen. Etwa Speicher für die Instanz zu allokieren, die Initialisierung, die Festlegung bestimmter Eigenschaften und so fort.

Jede Klasse besitzt mindestens eine Konstruktormethode, selbst wenn die Definition der Klasse keine solche Konstruktormethode explizit ausweist. In diesem Fall wird in Java eine Standardkonstruktormethode bereitgestellt, welche die Klasse von einer ihrer Superklassen vererbt bekommt (und wenn es von `java.lang.Object` ist). Es kann in einer Klasse mehrere Konstruktormethoden geben. Konstruktoren verhalten sich da wie andere Methoden und können überladen werden. Damit ist es möglich, mit einer Klasse Objekte auf vielfältige Art und Weise zu erstellen, indem mehrere Konstruktoren mit unterschiedlichen Parametern erstellt werden. Die Konstruktoren müssen sich wie andere Methoden nur innerhalb einer Klasse irgendwo in der Methodenunterschrift signifikant unterscheiden, damit sie überladen werden können. Die Methodenunterschrift bei Konstruktoren lässt dazu nicht viel Freiheiten, denn der Name des Konstruktors (ein Teil der Methodenunterschrift) ist zwingend zum Namen der Klasse identisch. Weiterhin dürfen Konstruktoren (obwohl sie Methoden sind) keine Rückgabeparameter (nicht einmal die Deklaration als `void` ist erlaubt) haben, weil sie ausschließlich dazu benutzt werden, eine Instanz der zugehörigen Klasse zurückzugeben.

Auch die Wahl der Modifier ist stark eingeschränkt. Entweder haben sie keinen Modifier (also die Voreinstellung freundlich), aber meist sind Konstruktoren als `public` deklariert. Sie dürfen nicht als `native, abstract, static, synchronized`, oder `final` deklariert werden. Es kann also nur Unterschiede in der Parameterliste geben.

Der Einsatz von Konstruktoren in Verbindung mit dem Schlüsselwort `new` vollzieht sich in zwei Schritten (welche auch in einer Quelltextzeile aufgeschrieben werden können).

- Über die Deklaration einer Instanzvariablen wird eine Variable eines bestimmten Typs mit einem Namen festgelegt.
- Über das Schlüsselwort `new` und einen Konstruktor dieser Klasse wird die konkrete Instanz erstellt, die dann der Variablen zugewiesen wird.

Die Syntax der Anweisung mit dem Operator `new` sieht folgendermaßen aus (in einer Quelltextzeile):

```
KlassenName instanzderKlasse =
 new KlassenName(optionale_parameter);
```

Dabei ist `KlassenName` die Klasse selbst und `instanzderKlasse` die Referenzvariable, die auf die Instanz der Klasse `KlassenName` verweist. Die Angabe `KlassenName()` – beachten Sie unbedingt die Klammern – auf der rechten Seite des Ausdrucks ist die Konstruktormethode. Die Parameter darin sind optional – je nachdem, ob die immer vorhandene – eventuell überschriebene – Standardkonstruktormethode (das bedeutet ohne Parameter) oder eine über Parameter spezifizierte Konstruktormethode aufgerufen wird. Anzahl und Typ der Argumente sind durch die konkrete Konstruktor-methode festgelegt.

Wenn Konstruktoren selbst innerhalb einer Klasse erstellt werden sollen, braucht bloß eine Methode mit gleichem Namen wie die Klasse dort notiert werden (ohne Rückgabewert!).

Warnungen

Konstruktoren lassen sich zwar überladen (was erst die polymorphe Funktionalität von Konstruktoren bewirkt), aber überschreiben geht rein technisch nicht, da sie immer den gleichen Namen wie die aktuelle Klasse haben müssen und deshalb nicht von einer Superklasse an die Subklasse vererbt, sondern in der Subklasse immer neu erstellt werden. Allerdings ist es auch bei Konstruktoren einer Subklasse möglich, die Konstruktoren der Superklasse mittels `super()` (beziehungsweise mit geeigneten Parametern) aufzurufen. Ein solcher Aufruf der Konstruktoren der Superklasse muss allerdings die erste Anweisung im Konstruktor der Subklasse sein.

Beispiele

Die Verwendung des `new`-Operators mit drei verschiedenen Konstruktoren:

```
class MeineKlasse
{
 public static void main (String args[])
 {
  Date datum1, datum2, datum3;
  datum1 = new Date();
```

```
  datum2 = new Date(97, 8, 8);
  datum3 = new Date("April, 17, 1963, 3:24, PM");
...
 }
}
```

Die Deklaration einer Variablen und die konkrete Erzeugung einer Instanz in einer Zeile:

```
class MeineKlasse
{
 public static void main (String args[])
 {
...
  Date datum1 = new Date();
...
 }
}
```

null

Das Null-Literal.

Beschreibung

Der Nulltyp in Java hat einen Wert, die `null`-Referenz, welche über das Literal `null` dargestellt wird. Ein Null-Literal ist immer vom Typ `null`.

Anwendung

In Java haben alle Referenzdatentypen den Defaultwert `null`. Dieses reservierte Schlüsselwort weist darauf hin, dass eine Variable nicht auf ein Objekt verweist. Man kann dies in Vergleichen nutzen, ob ein konkretes Objekt zur Verfügung steht oder nicht. Ansonsten kann das Schlüsselwort dann verwendet werden, wenn eine Syntax auf ein Objekt angewiesen ist, aber für die konkrete Funktionalität kein besonderes Objekt benötigt wird. Desgleichen wird bei Ausnahmen oft der Wert `null` zurückgegeben.

Beispiele

Vergleich, ob eine Referenzvariable auf ein Objekt verweist:

```
if (MeineObjektVariable == null);
 {
 }
```

package

Der Beginn einer Paketdeklaration.

Beschreibung

Mit diesem Schlüsselwort wird festgelegt, dass es sich um die Deklaration eines neuen Paketes handelt oder der nachfolgende Code einem bestehenden Paket zugeordnet werden soll.

Anwendung

Das Paketkonzept von Java dient dazu, mehrere Klassen zu einem Paket über die Anweisung `package` zusammenzufassen. Durch die Anweisung `import` werden einzelne Pakete dann in einem Programm verfügbar gemacht oder man greift per Punktnotation über die Paketstruktur auf eine Klasse beziehungsweise ein Klassenelement zu.

Java-Klassen und Schnittstellen können frei zu Paketen zusammenfasst und gruppiert werden. Eine Java-Datei wird ganz einfach einem Paket beziehungsweise einem bestehenden Paket zugeordnet, wenn am Anfang der Datei als erste gültige Anweisung (auch vor der ersten Klassendefinition, aber abgesehen von Kommentaren) das Schlüsselwort `package`, gefolgt von dem Namen des Paketes und einem Semikolon, steht. Anschließend werden wie gewohnt die Klassen oder Schnittstellen definiert.

Wenn sich innerhalb einer mit `package` eingeleiteten Quelldatei mehrere Klassen- beziehungsweise Schnittstellendefinitionen befinden, dann werden alle Klassen/ Schnittstellen dem angegebenen Paket zugeordnet.

Pakete werden in der Regel so organisiert wie die Vererbungshierarchie. Die Java-Klassenbibliothek ist selbst so organisiert. Die Ähnlichkeit der Paketstruktur zum Dateisystem auf dem Rechner ist nicht ganz zufällig, da sich oft einzelne Klassen und Schnittstellen beziehungsweise logisch zusammengehörende Codestrukturen in einer eigenen Datei befinden, welche wiederum in eine zu dem Paketnamen passenden Verzeichnisstruktur eingebunden ist. Die Paketstruktur spiegelt also in der Regel eine Verzeichnisstruktur auf Ebene des Betriebssystems wider.

Bei den Namenskonventionen für Pakete gelten die üblichen Regeln für Token, wobei die erste Ebene der Namenshierarchie oft den eindeutigen Namen des Paketentwicklers (etwa eine Firma) spezifiziert.

In der Vergangenheit bezeichnete `package` auch einen Zugriffspezifizierer. Damit wurde festgelegt, dass bestimmte Elemente paketweit sichtbar waren. Dies ist nicht mehr notwendig, denn diese freundliche (heutige Bezeichnung) Einstellung ist Defaultwert.

Warnungen

- Wenn Sie in einem Paket selbst ein anderes Paket importieren wollen, muss die `import`-Anweisung für das zu importierende Paket unmittelbar nach der `package`-Anweisung für das neu zu deklarierende Paket stehen.
- Wenn Sie selbst ein Package erstellen, sollten sämtliche Dateien, welche diesem Paket zugeordnet sind, in einem Verzeichnis stehen, welches mit dem Namen des Paketes identisch ist (beachten Sie dabei den Suchpfad des Compilers – nahezu immer geht ein Unterverzeichnis von dem Verzeichnis aus, wo die importierende Datei steht). Der Compiler wird beim Übersetzen einer Datei mit einem `import`-Befehl, der auf das Paket verweist, sonst unter Umständen Probleme haben. Sie erhalten dann eventuell Fehlermeldungen der Art:
  Package ... not found in import.
  oder
  error: File .\....class does not contain type ... as expected, but type .... Please remove the file, or make sure it appears in the correct subdirectory of the class path.
  ...: Superclass ... of class ... not found.
  public class ... extends ...

Beispiele

Eine Paketdeklaration:

```
package MeinPaket;
public class MeineKlasseImPaket

...
}
```

Tips

Wenn die `package`-Anweisung am Beginn einer Quelldatei fehlt, wird die Klasse einem voreingestellten Paket ohne Namen (einem sogenannten anonymen Paket) zugeordnet. Die Klassen dieses Paketes können dann direkt von allen Klassen importiert werden, welche im gleichen Verzeichnis stehen (und nur von diesen).

Verwandte Befehle

```
class
interface
import
```

private

Ein Zugriffsspezifizierer für Klassenmitglieder.

Beschreibung

Die höchste auf ein Klassenmitglied (eine Variable oder eine Methode) anwendbare Sicherheitsstufe.

Anwendung

Über den einem Mitglied einer Klasse vorangestellten Zugriffsspezifizierer `private` wird festgelegt, dass dieses Element ausschließlich innerhalb des Körpers der umgebenden Klasse verfügbar ist. Sogar eine Subklasse dieser Klasse kann auf eine private Methode oder Variable nicht zugreifen.

Warnungen

Eine Klasse selbst kann nicht als `private` deklariert werden. Sie ist immer mindestens im gesamten Paket sichtbar. Wenn eine Klasse als `private` deklariert wird, erzeugt der Compiler eine Fehlermeldung der Art:

The type type ... can't be private. Package members are always accessible within the current package.

Beispiele

Eine als privat deklarierte Methode:

```
private void geheimeMethode()
```

Eine als privat deklarierte Variable:

```
private final String Antwort = "Zweiundvierzig";
```

Verwandte Befehle

```
protected
public
package
```

protected

Ein Zugriffsspezifizierer für Klassenbestandteile.

Beschreibung

Über den vorangestellten Zugriffsspezifizierer `protected` wird eine Einschränkung der Sichtbarkeit eines Klassenmitglieds erreicht.

Anwendung

Der Zugriffsspezifizierer `protected` kann auf Methoden und Variablen angewandt werden. Er legt fest, dass ein Mitglied einer Klasse innerhalb des Paketes verfügbar ist, nicht aber außerhalb, es sei denn, es handelt sich um eine Subklasse. Subklassen haben vollen Zugriff auf alle als `protected` deklarierten Elemente ihrer Superklassen (auch wenn sich Subklasse und Superklasse in verschiedenen Paketen befinden).

Warnungen

- Im Prinzip könnten auch Klassen als `protected` deklariert werden. Als `protected` deklarierte Klassen könnten sich aber als ziemlich gefährlich erweisen, weil Subklassen damit vollen Zugriff auf ihre Superklasse hätten. Dies lässt man in Java nicht zu. Wenn Klassen so deklariert werden, wird der Compiler einen Fehler melden (*Class or interface declaration expected.*). Aber auch die unkritische Deklaration von Variablen als `protected` kann fatal sein. Falls in einer Subklasse auf Variablen der Superklasse zugegriffen wird und diese in der Subklasse konkret verwendet werden, kann eine spätere Veränderung in der Superklasse fatale Folgen haben. Die Philosophie der OOP fordert deshalb ja explizit die Datenkapselung, also das Verstecken von allen Elementen eines Objektes, die nicht für die konkrete Funktionalität notwendig sind (auch das Verstecken von unnötigen oder kritischen Informationen in einer Superklasse gegenüber potentiellen Subklassen – insbesondere, wenn diese nicht von dem gleichen Programmierer erstellt werden und/oder sie sich in jedem Fall in einem fremden Paket befinden). Das (gezielte) Deklarieren von Methoden (oder auch in einzelnen Fällen von bestimmten Variablen, die notwendige Informationen über das Objekt beinhalten) als `protected` macht dagegen viel mehr Sinn, weil dann nur gezielt die Informationen nach außen gelassen werden (bei sinnvoller Programmierung), die für die gewünschte Funktionalität notwendig sind.
- Ein Kompilierfehler kann unter Umständen darauf zurückzuführen sein, dass versucht wird, auf eine Methode außerhalb des sichtbaren Anwendungsbereiches einer geschützten Methode zuzugreifen. Die dann entstehende Fehlermel-

dung ist leider missverständlich, denn sie weist nicht darauf hin, dass versucht wird, auf eine geschützte Methode zuzugreifen. Statt dessen sieht sie etwa so aus:

No method matching speichereGeheimzahl() found in class java.xyz.jhg.

- Der Grund ist der, dass die geschützten Methoden vor nichtprivilegierten Klassen versteckt werden. Wenn deshalb eine Klasse kompiliert wird, welche die Sicherheitsbestimmungen nicht erfüllt, werden Methoden dieser Art vor dem Compiler versteckt. Ähnliche Fehlermeldungen entstehen genauso, wenn Sie versuchen, außerhalb Ihrer Zugriffsrechte auf eine private oder freundliche Methode zuzugreifen, oder wenn Sie versuchen, von einer nichtprivilegierten Klasse auf ein Feld zuzugreifen.
- Das in der Sprache C/C++ ebenfalls vorhandene Schlüsselwort `protected` ist von der Funktionalität nicht vollkommen identisch mit dem Java-Schlüsselwort `protected` .

Beispiele

Eine `protected` deklarierte Methode:

```
protected void speichereGeheimzahl()
```

Eine `protected` deklarierte Variable:

```
protected int test
```

Verwandte Befehle

`private`
`public`
`package`

public

Ein allgemeiner Zugriffsspezifizierer.

Beschreibung

Über den vorangestellten Zugriffsspezifizierer `public` wird eine Erweiterung der Sichtbarkeit einer Klasse, einer Schnittstelle, eines Klassenmitglieds oder eines Schnittstellenmitglieds erreicht.

Anwendung

Der Zugriffsspezifizierer `public` kann auf Klassen, Schnittstellen, Klassenmitglieder oder Schnittstellenmitglieder angewandt werden. Er wird in der Deklaration vorangestellt und legt fest, dass diese Elemente überall verfügbar sind.

Warnungen

Als `public` deklarierte Klassen und Schnittstellen müssen zwingend in einer Datei gespeichert werden, die identisch zu dem Klassen- beziehungsweise Schnittstellennamen ist (plus die Erweiterung `.java`). Daraus folgt zwingend, dass jeweils nur eine als `public` deklarierte Klasse oder Schnittstelle in einer Datei gespeichert werden kann.

Beispiele

Eine `public` deklarierte Methode:

```
public void speichereNichtgeheimzahle()
```

Eine `public` deklarierte Klasse:

```
public class MeineKlasse()
```

Eine `public` deklarierte Variable:

```
public int test
```

Verwandte Befehle

```
private
protected
package
```

return

Eine Sprunganweisung, die im Allgemeinen einen Wert zurückliefert.

Beschreibung

Sprunganweisungen geben die Steuerung entweder an den Anfang oder das Ende des derzeitigen Blocks, oder aber an bezeichnete Anweisungen weiter. Eine `return`-Anweisung gibt die Kontrolle an den Initiator der Methode, des Konstruktors oder des statischen Initialisators zurück. Außerdem wird bei entsprechender Syntax ein Rückgabewert an den Aufrufer übergeben.

Anwendung

Die `return`-Anweisung kann sowohl nur den Kontrollfluss an einen Aufrufer zurückgeben (bei einer als `void` deklarierte Methode, indem einfach das Schlüsselwort `return` ohne weitere Angaben notiert wird), als auch dabei einen Parameter (den Rückgabewert) an den Aufrufer übergeben. Der Rückgabewert muss dem Typ der Methode entsprechen und wird hinter dem Schlüsselwort `return` notiert. Dabei kann er mit oder ohne Klammern notiert werden.

Die `return`-Anweisung wird im Zusammenhang mit einem statischen Initialisator automatisch ausgelöst, wenn ein Compilerfehler auftritt.

Warnungen

- Wenn `return` innerhalb einer `try-catch`-Anweisung ausgelöst wird und es einen `finally`-Zweig gibt, wird dieser Teil immer noch ausgeführt, bevor die Kontrolle weitergegeben wird.
- Sofern eine Methode nicht explizit als `void` deklariert ist, muss per `return` ein Rückgabewert ausgeworfen werden. Andernfalls gibt es einen Compilerfehler. Im `void`-Fall ist es freigestellt, ob mit `return` (ohne Rückgabewert) der Kontrollfluss zurückgegeben wird oder er automatisch zurückkehrt.
- Wenn nach einer `return`-Anweisung noch Code steht, der nie erreicht werden kann (etwa weil die `return`-Anweisung in jedem Fall durchgeführt wird), wird der Compiler einen Fehler melden. Die `return`-Anweisung sollte deshalb immer am Ende eines Anweisungsblocks stehen.

Beispiele

Eine Methode mit Rückgabe eines `int`-Wertes:

```
public static int zurueck()
{
 return 42;
}
...
public static void main (String args[])
{
 int i=zurueck();
 ...
}
```

Eine als `void` deklarierte Methode mit `return`-Anweisung – ohne weiteren Parameter:

```
public static void eins()
{
 ...
 return;
}
```

Verwandte Befehle

`break`
`continue`
`throw`

short

Ein Datentyp (Ganzzahltyp).

Beschreibung

Der Datentyp `short` dient zur kurzen Darstellung von Ganzzahlwerten (ganzzahliges Zweierkomplement mit Vorzeichen). Er hat eine Länge von 16 Bit. Defaultwert einer Variablenn dieses Typs ist 0. Dieser Datentyp kann die Ganzzahlwerte von (– 2 hoch 15 =) -32.768 bis (+ 2 hoch 15 – 1 =) +32.767 darstellen.

Anwendung

Diesen Datentyp wird man in Java nicht so häufig wie den Datentyp `int` finden, obwohl er ähnliche Anwendungsmöglichkeiten bietet. Grundsätzlich wird mit diesem Schlüsselwort einer Variablen ein Datentyp oder einer Methode ein Rückgabetyp zugewiesen. Eine Anwendung von `short`-Datentypen anstelle von `int`-Typen macht immer dann Sinn, wenn Speicherplatz gespart werden soll (etwa bei großen Datenstrukturen wie multidimensionalen Arrays als Zählindex) und der Wertebereich klein genug bleibt.

Warnungen

- Zwei ganzzahlige Datentypen (`byte`, `short`, `int` oder `long`) als Operanden ergeben immer einen ganzzahligen Datentyp als Ergebnis. Dabei kann als Datentyp des Ergebnisses (auch bei der Operation von zwei `short`-Datentypen miteinander!) immer nur ein Datentyp `int` oder `long` entstehen. Defaultmäßig wird das Ergebnis vom Typ `int` sein. Der Datentyp long entsteht dann und nur dann, wenn einer der beiden Operanden bereits vom Datentyp `long` war, oder das Ergebnis von der Größe her nur in einem Datentyp `long` dargestellt werden kann. Um ein Ergebnis vom Typ `short` (oder auch `byte`) als Ergebnis zu bekommen, muss es explizit festgelegt werden.
- Der in Java verwendete Unicode-Standard (Darstellung in 16 Bit) ermöglicht den Gebrauch von Alphabeten vieler verschiedener Sprachen. So dargestellte Zeichenvariablen können Operanden in jeder ganzzahligen Operation sein, und werden dabei wie ganze 16-Bit-Zahlen ohne Vorzeichen behandelt werden. Recht offensichtlich ist die Tatsache, dass unter Umständen einige Informationen verloren gehen, wenn Zeichen in einen kleineren Typ (ein Byte) umgewandelt werden. Dies betrifft beispielsweise den Han-Zeichensatz (Chinesisch, Japanisch oder Koreanisch). Aber oft wird übersehen, dass auch die Umwandlung von einer Variablen vom Datentyp `char` in eine Variable vom Datentyp `short` zu Informationsverlust führen kann – trotz gleicher Länge der Datentypen. Der Grund ist die Darstellung von `short` als ganzzahliges Zweierkomplement mit Vorzeichen, während `char` vorzeichenlos wie ganze 16-Bit-Zahlen (Werte 0 bis 65535) entsprechend der Unicode-Kodierung behandelt wird.

Beispiele

Eine Methodendeklaration mit einem Rückgabewert vom Typ `short`:

```
short test();
```

Eine Variable vom Typ `short`:

```
short test;
```

Verwandte Befehle

Die übrigen Datentypen von Java:

```
boolean
byte
char
double
float
int
long
```

static

Ein Modifier für Methoden und Variablen.

Beschreibung

Das Schlüsselwort deklariert Methoden oder Variablen als statisch. Dies bedeutet, sie werden damit zu Klassenmethoden beziehungsweise Klassenvariablen.

Anwendung

Java unterscheidet zwischen den Methoden und Eigenschaften einer spezifischen Instanz einer Klasse und Methoden und Eigenschaften der Klasse selbst. Die Klassenmethoden beziehungsweise Klassenvariablen liegen außerhalb der konkreten Objekte und sind direkter Bestandteil der jeweiligen Klasse. Sie werden über das Metaklassenkonzept von Java in das streng objektorientierte Javakonzept eingebunden. Klassenmethoden beziehungsweise Klassenvariablen sind immer global.

Deklariert werden Klassenmethoden respektive Klassenvariablen wie Instanzmethoden beziehungsweise Instanzvariablen (also Methoden und Variablen, die auch in der konkreten Instanz sichtbar sind), nur ergänzt durch das Voranstellen des Modifiers `static`.

Eine der wichtigsten Anwendungen des Modifiers `static` findet man bereits am Einstiegspunkt für jedes Java-Programm. Jedes Java-Programm benötigt eine Methode namens `public static void main(String args[])`.

Warnungen

- Nichtstatische Methoden können zwar mit statischen Variablen arbeiten, statische Methoden demgegenüber nur mit statischen Variablen und anderen statischen Methoden. Es ist ein klassischer Fehler, wenn aus einer statischen Methode eine nichtstatische Variable oder eine nichtstatische Methode referenziert werden soll. Der Compiler erzeugt dabei eine Fehlermeldung der Art *Can't make a static reference to nonstatic...*
- Klassenmethoden können in einer Subklasse nicht durch eine Instanzmethode überschrieben werden. Falls Sie dies tun, erhalten Sie eine Fehlermeldung der Art
  ....java:...: The instance method ... declared in class ... cannot override the static method of the same signature declared in class .... It is illegal to override a static method. Ein Überschreiben in der Subklasse durch eine weitere Klassenmethode ist jedoch machbar. Ein Überladen einer Klassenmethode durch eine Instanzmethode innerhalb derselben Klasse in dem Sinne, dass sich die beiden Methodenunterschriften nur durch den vorangestellten Modifier unterscheiden (also nur zusätzlich `static` für die Klassenmethode), ist ebenfalls nicht

möglich. Es wird durch den Compiler eine Fehlermeldung der Art *....java:7: Duplicate method declaration: ...* ausgelöst.

- Alle in Schnittstellen deklarierten Variablen sind `static`, auch wenn dies nicht explizit in der Deklaration angegeben ist. Dagegen dürfen Methoden bei der Deklaration in einer Schnittstelle nicht als `static` deklariert werden.
- Konstruktoren dürfen nie als `static` deklariert werden.

Beispiele

Eine Klassenmethode

```
public static void main(String args[])
```

Eine statische Variable:

```
static int test;
```

Verwandte Befehle

```
public
native
abstract
synchronized
final
private
protected
```

super

Zugriff auf eine Superklasse.

Beschreibung

Über das Schlüsselwort `super` kann aus einer Subklasse auf Elemente ihrer Superklasse zugegriffen werden.

Anwendung

Um auf ein Element einer Superklasse innerhalb einer abgeleiteten Subklasse zugreifen zu können, steht das Schlüsselwort `super` zur Verfügung. Über die Punktnotation mit vorangestelltem `super` ist es ganz einfach möglich, Methoden oder Variablen aus der Superklasse direkt anzusprechen. Dies ist insbesondere dann von Bedeutung, wenn ein Element in der Subklasse überschrieben wurde und man sowohl die überschriebene Variante als auch gleichzeitig die Originalversion der Superklasse nutzen möchte. Um eine Methode in einer Superklasse aufzurufen, wird also die folgende Syntax verwendet:

```
super.methodenname(Parameter);
```

Für Variablen gilt dies analog.

Das Schlüsselwort `super` wird in Java noch in einem erweiterten Zusammenhang verwendet – um einen Konstruktor der Superklasse innerhalb der Subklasse direkt aufzurufen. Dies könnte im Prinzip analog der obigen Syntax (`super.konstruktorname(Parameter)`) erfolgen, jedoch ist bei Konstruktoren der Methodenname überflüssig, denn der muss sowieso identisch zum Klassennamen sein. Deshalb kann bei Konstruktoren die folgende verkürzte Syntax verwendet werden:

```
super(Parameter);
```

Warnungen

- Das Schlüsselwort `super` kann nur im Körper einer nichtstatischen Methode verwendet werden.
- Der Aufruf eines Konstruktors einer Superklasse über `super()` (eventuell mit geeigneten Parametern) kann nur im Konstruktor einer Subklasse erfolgen und muss dann als erste Anweisung stehen.

Beispiele

Zugriff auf die Methode einer Superklasse

```
super.meineMethode();
```

Aufruf eines Konstruktors der Superklasse mit Parameter:

```
super(5);
```

Verwandte Befehle

new
this

switch

Eine Auswahlanweisung. Der Beginn der `switch-case-default`-Struktur.

Beschreibung

Über eine Auswahlanweisung wird einer von mehreren möglichen Kontrollflüssen ausgesucht.

Anwendung

Über die `switch`-Anweisung wird einer von mehreren möglichen Kontrollflüssen ausgesucht. Die `switch`-Anweisung kann nur in Verbindung mit der `case`-Anweisung eingesetzt werden, welche die möglichen verschiedenen Anweisungsblöcke mit Unteranweisungen einleitet.

An welche Anweisung dieser Anweisungsblöcke innerhalb der `switch`-Anweisung der Kontrollfluß weitergereicht wird, hängt von der Übereinstimmung zwischen dem Wert des `case`-Ausdrucks und dem in der `switch`-Anweisung getesten Wert ab. Es wird immer die erste Anweisung nach einer `case`-Bezeichnung ausgeführt, welche denselben Wert wie der getestete Ausdruck hat. Wenn es keine entsprechenden Werte gibt, wird die erste Anweisung hinter der letzten `case`-Bezeichnung – unter Umständen ein optionaler `default`-Zweig – ausgeführt. Wenn dieser optionale `default`-Zweig nicht vorhanden ist, wird die erste Anweisung nach dem `switch`-Block ausgeführt. Die `switch`-Ausdrücke (und damit auch die `case`-Bezeichnungskonstanten) müssen alle vom Typ `byte`, `short`, `char` oder `int` sein.

Warnungen

- Es dürfen keine zwei `case`-Bezeichnungen im gleichen `switch`-Block denselben Wert haben.
- Der Kontrollfluss wird nach einer Übereinstimmung mit einer `case`-Bezeichnung und nachfolgender Abarbeitung nicht automatisch aus der `switch`-Struktur zurückgegeben. Alle nachfolgenden `case`-Blöcke werden ebenfalls abgearbeitet! Dies kann man mit einer `break`-Anweisung am Ende eines `case`-Blocks unterbinden.

Beispiele

Ein `switch-case`-Konstrukt ohne `default`-Zweig und mit `break`-Anweisungen am Ende jedes `case`-Blocks:

```
short test; // Eine numerische Testvariable Typ short
...
switch (test)
  {
  case 0:
   ...
   break;
  case 1:
   ...
   break;
  case 2:
   ...
   break;
  case 3:
   ...
   break;
}
...
```

Ein `switch-case-default`-Konstrukt ohne `break`-Anweisungen am Ende jedes `case`-Blocks mit `char`-Testvariable:

```
char test; // Eine numerische Testvariable
...
switch (test)
  {
  case A:
   ...
  case B:
   ...
  case C:
   ...
  default:
   ...
}
...
```

Tips

Um den Kontrollfluss nach einer Übereinstimmung mit einer `case`-Bezeichnung und nachfolgender Abarbeitung aus der `switch`-Struktur ohne Abarbeitung der nachfolgenden `case`-Blöcke zurückgegeben, muss eine `break`-Anweisung am Ende eines `case`-Blocks folgen. Natürlich können Sie dieses Verhalten auch gezielt einsetzen, um durch Auslassen von `break`-Anweisungen den Durchlauf von mehreren Blöcken zu erreichen.

Verwandte Befehle

`break`
`case`
`default`
`if`
`if-else`

synchronized

Eine Synchronisationsanweisung für verschiedene Threads.

Beschreibung

Über eine Synchronisationsanweisung werden im Rahmen von Multithreading Methoden und Blöcke vor gleichzeitiger Verwendung geschützt.

Anwendung

Die einer Methode vorangestellte Synchronisationsanweisung `synchronized` markiert eine Methode oder einen Block mit Anweisungen, um eine Datenverletzung im Rahmen von Multithreading zu verhindern. Bei parallel laufenden Threads kann es sonst passieren, dass diese gleichzeitig versuchen, in kritischer oder sonst nicht gewollter Form auf dieselben Daten zuzugreifen (etwa ein Schreibzugriff).

Um solche potentiellen Datenverletzungsprobleme zu vermeiden, werden die betreffenden Methoden beziehungsweise Codeblöcke allesamt als `synchronized` deklariert. In einem bestimmten Objekt dürfen keine zwei synchronisierten Methoden respektive Codeblöcke gleichzeitig laufen. Wenn eine Methode beziehungsweise ein Codeblock aufgerufen wird, wird eine Art »Schloss« für das Objekt erstellt. Wenn dann eine weitere synchronisierte Methode beziehungsweise ein synchronisierter Codeblock aufgerufen wird, muss die zweite Anweisungsfolge solange warten, bis die erste fertig ist und das Schloss für dieses Objekt wieder geöffnet wurde.

Es gibt einen entscheidenden Unterschied zwischen synchronisierten Methoden und synchronisierten Blöcken. Synchronisierte Methoden verschließen die derzeitige Klasse, während synchronisierte Blöcke nur die spezifizierte Methode während der Ausführung verschließen.

Warnungen

- Konstruktoren dürfen nicht als `synchronized` deklariert werden.
- Bei der Deklaration einer Methode in einer Schnittstelle darf `synchronized` nicht verwendet werden.

Beispiele

Zwei synchronisierte Methoden:

```
class MeineKlasse
{
Zugriff oeffneSchnittstelle;
synchronized void oeffneSchnittstelle
{
...
online(oeffneSchnittstelle.sende[1]);
online(oeffneSchnittstelle.sende[2]);
}
synchronized void oeffneSchnittstelleNochmal
{
...
}
}
```

Tips

Die Verwendung des Schlüsselworts für eine Methode macht ebenfalls Sinn, wenn sie von verschiedenen Threads gleichzeitig aufgerufen werden kann. Das Problem ist jedoch, dass solche Synchronisierungen den Programmablauf verlangsamen können. Deshalb sollte das Schlüsselwort nur zum Einsatz kommen, wenn dies unbedingt erforderlich ist.

Verwandte Befehle

```
transient
volatile
```

this

Eine Variable zum Zugriff einer Klasse beziehungsweise Klasseninstanz auf sich selbst.

Beschreibung

Damit eine Klasse beziehungsweise eine Instanz auf sich selbst zugreifen kann, stellt Java das Schlüsselwort `this` zur Verfügung.

Anwendung

Für die Anwendung des Schlüsselwortes `this` gibt es im Allgemeinen zwei Situationen:

- Es gibt in einer Klasse zwei Variablen mit gleichem Namen – eine gehört zu der Klasse, die andere zu einer spezifischen Methode in der Klasse. Die Benutzung der Syntax `this.VariablenName` ermöglicht es auch innerhalb der spezifischen Methode in der Klasse, auf diejenige Variable zuzugreifen, die zu der Klasse gehört. Das Schlüsselwort `this` kann weggelassen werden, wenn auf Elemente der Klasse verwiesen wird und es im lokalen Bereich kein Element mit dem gleichen Namen gibt. In diesem Fall wird `this` implizit in diesen Referenzen verwendet.
- Eine Klasse muss sich selbst als Parameter für eine Methode weiterreichen beziehungsweise der Rückgabetyp einer Methode ist eine Objektinstanz der Klasse. Das Schlüsselwort `this` kann an jeder Stelle verwendet werden, an der das Objekt erscheinen kann. Etwa als Argument einer Methode, als Ausgabewert, oder in Form der Punktnotation zum Zugriff auf eine Instanzvariable.

Warnungen

Das Schlüsselwort `this` kann nur im Körper einer nichtstatischen Methode verwendet werden.

Beispiele

Zugriff innerhalb einer Methode auf eine Variable der Klasse:

```
class MeineKlasse
{
int x = 0;
void meineMethode(int x)
{
...
 this.x = x;
}
...
}
```

Verwandte Befehle

`super`

throw

Eine Sprunganweisung im Rahmen der Ausnahmebehandlung von Java.

Beschreibung

Die `throw`-Sprunganweisung wirft eine Ausnahme aus und gibt die Steuerung zurück an die aufrufende Methode. Sie ist ein wichtiger Bestandteil des Ausnahmemechanismus von Java.

Anwendung

In Java ist es möglich, durch die `throw`-Anweisung eine Laufzeitausnahme des Programms zu erzeugen. Dies bedeutet, dass der normale Programmablauf durch eine Ausnahme unterbrochen wird, die zuerst von dem Programm behandelt werden muss, bevor der normale Programmablauf weitergeht. Die Laufzeitausnahme verwendet ein Objekt als Argument. Dieses Objekt, welches hinter der Anweisung `throw` steht, bezeichnet einen Referenzausdruck, der gewöhnlich von der Klasse `Exception` abgeleitet sein muss.

Wenn ein Java-Interpreter auf eine `throw`-Anweisung stößt, passieren zwei Dinge:

- Die `throw`-Anweisung wird in Java immer zum direkten Auslösen einer Exception verwendet und übergibt als Argument eine Instanz der Ausnahme, die ausgelöst werden soll, an die nächsthöhere Hierarchieebene. Die `throw`-Anweisung ist vom Programmablauf her gut mit der `break`-Anweisung vergleichbar. Was in dem Block danach folgt, wird nicht mehr ausgeführt.
- Grundsätzlich wird beim Auftreten einer Ausnahme die Ausführung eines Programms solange unterbrochen, bis eine passende `catch`-Anweisung mit einem Formalparameter gefunden wird (auch über verschiedene Hierarchieebenen hinweg), der die Superklasse des Typs in der `throw`-Anweisung ist.

Es können mit einer `throw`-Anweisung sowohl eine der von Java in seinen Standardbibliotheken vorgefertigten Ausnahmen für fast alle wichtigen Standardsituationen (etwa bei Dateioperationen auf Basis der `IOException`) ausgeworfen werden, als auch eine im Programm selbst definierte Ausnahme. Um eine solche Ausnahme zu erstellen, muss nur eine Unterklasse von `Throwable` oder eine ihrer Unterklassen wie `Exception` implementiert werden.

Warnungen

- Der `finally`-Teil jeder zum Zeitpunkt des Aufrufs der `throw`-Anweisung aktiven `try`-Anweisung wird immer ausgeführt.
- Die `throw`-Sprunganweisung darf nicht mit der `throws`-Anweisung verwechselt werden (trotz einer gewissen Verwandtschaft). Die beiden Anweisungen sind freilich eng miteinander verbunden.

Beispiele

Eine Methode, welche eine selbstdefinierte Ausnahme auswirft.

```
static void meineMethode() throws MeineAusnahme
{
// Erzeugen eines Ausnahmeobjektes mit einem Konstruktur
MeineAusnahme m = new MeineAusnahme();
...
throw m; // Auswerfen der Ausnahme
}
```

Verwandte Befehle

```
break
catch
continue
return
throws
try
finally
```

throws

Eine optionale Erweiterung einer Methodendeklaration zur Ausnahmendeklaration.

Beschreibung

Java realisiert das globale Handling von Ausnahmen über eine optionale Erweiterung einer Methodendeklaration um die `throws`-Anweisung. Damit wird eine Liste mit den Ausnahmen deklariert, welche von der Methode ausgeworfen werden können.

Anwendung

Wenn innerhalb einer Java-Methode kritische Aktionen durchgeführt werden, welche zur Laufzeit zu nicht genau vorhersehbaren Problemen führen können (etwa Dateizugriffe auf Datenträger, von deren Existenz nicht zuverlässig ausgegangen werden kann – beispielsweise Disketten), werden diese am besten über eine Ausnahme abgefangen.

Die Deklarationen von Methoden sehen generell folgendermaßen aus:

```
[Zugriffsspezifizierer] [modifier] return_wert namedermethode([Parameter])
[throws] [ExceptionListe]
```

Dabei ist alles in eckigen Klammern optional.

Wenn nun eine Deklaration einer oder mehrerer Ausnahme(n) mittels der `throws`-Anweisung erfolgt, ist das als eine reine Dokumentation zu verstehen, die allerdings weitreichende Konsequenzen hat.

Die Dokumentation ist eine Art Vertrag zwischen der kritischen Methode und demjenigen, der die Methode verwendet. Von Seiten der kritischen Methode werden alle gefährlichen Dinge, welche bei der Methode auftreten können, mittels der `throws`-Anweisung spezifiziert (etwa die Ausnahmen, welche die Methode explizit per `throws`-Anweisung auswirft). Als Anwender der Methode müssen Sie sich darauf verlassen, dass diese Angaben die Methode korrekt charakterisieren. Auf Seiten des Anwenders verpflichtet die Verwendung von Ausnahmen in einer Methodendeklaration den Anwender, diese Ausnahme(n) zu behandeln. Der Java-Compiler wird Source, welcher eine kritische Methode verwendet, nur übersetzen, wenn die ausgeworfene(n) Ausnahme(n) an irgendeiner Stelle im Programm behandelt werden. Dabei muss man zwischen zwei Fällen unterscheiden:

- Eine Dokumentation einer Ausnahme durch die `throws`-Anweisung verpflichtet zu einer expliziten Behandlung durch den Anwender. Dies erfolgt in der Regel über eine die kritische Methode umgebende `try-catch`-Struktur (nicht unbedingt direkt in der aufrufenden Hierarchieebene).
- Es müssen nicht alle denkbaren Fehler und Ausnahmen explizit aufgelistet werden. Die Javastandardklasse `Throwable` besitzt zwei umfangreiche Subklassen,die

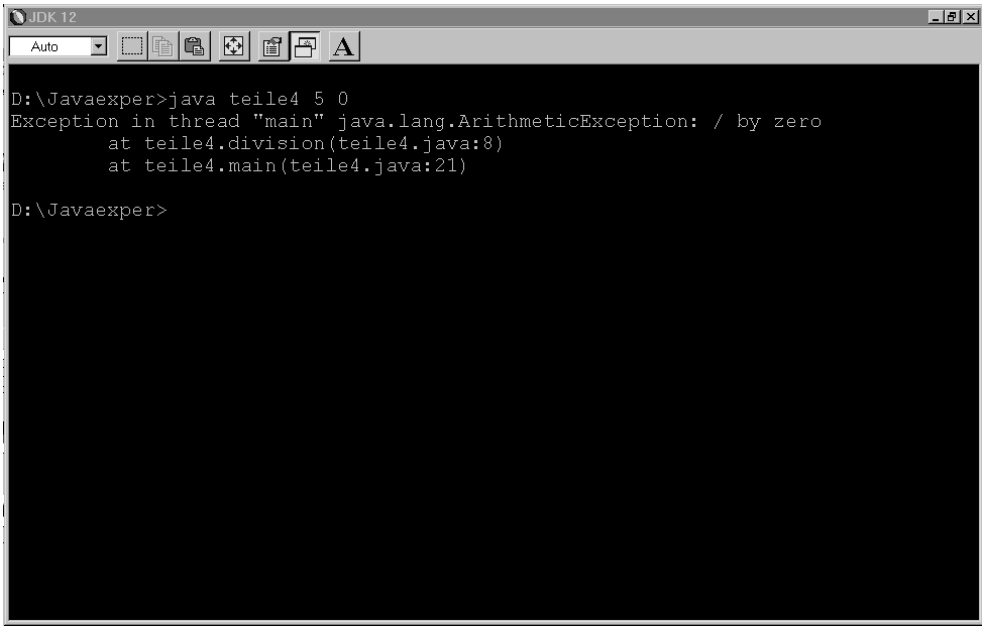
Klassen `Exception` und `Error`. In den beiden Ausnahmeklassen sind die wichtigsten Ausnahmen und Fehler der Lava-Laufzeitbibliothek bereits enthalten. Diese beiden Klassen bilden zwar zwei getrennte Hierarchien, werden aber ansonsten gleichwertig als Ausnahmen behandelt. Sämtliche Ausnahmen der Klasse `Error` und ihrer Subklasse `RuntimeError` müssen nicht explizit durch den Anwender der Methode abgefangen werden. Dies geschieht automatisch. In die Ausnahmen dieser Klassen fallen unter anderem Situationen wie Speicherplatzmangel oder `StackOverflowError`. In der Dokumentation von Java sind hunderte von Standardausnahmen dokumentiert, für die diese Aussage gilt. Diese Standardproblemfälle sollten, wenn sie im Rahmen der Verwendung in einer aufrufenden Methode auftreten können, auch gar nicht mehr explizit mit der `throws`-Anweisung dokumentiert werden, denn damit wird ein Aufrufer der Methode gezwungen, diese explizit zu handhaben (was sonst wie gesagt automatisch über die Java-Umgebung gehandelt wird). Selbstverständlich bedeutet die automatische Behandlung dieser Standardausnahmen auf Systemebene nicht, dass die Ausnahmen nicht auf freiwilliger Basis explizit behandelt werden können (etwa um qualifizierte Fehlermeldungen zu erzeugen).

Es gibt eigentlich nur fünf sehr allgemeine Typen von Ausnahmen, welche in einer `throws`-Klausel aufgelistet werden müssen und damit von einer anwendenden Struktur behandelt werden müssen. Man kann dies aus der Beschreibung der Klasse `java.lang` entnehmen. Es sind dies:

- `ClassNotFoundException`
- `IllegalAccessException`
- `InstantiationException`
- `InterruptedException`
- `NoSuchMethodException`

Wenn nun in einer mit der `throws`-Erweiterung gekennzeichneten Methode eine Ausnahme auftritt, wird die aufrufende Methode nach einer solchen Ausnahmebehandlungsstruktur durchsucht. Enthält die aufrufende Methode eine Ausnahmebehandlungsstruktur, wird mit dieser die Ausnahme bearbeitet; ist dort keine Routine vorhanden, wird deren aufrufende Methode duchsucht und so fort. Das Spiel geht solange weiter, bis eine Ausnahmebehandlungsmethode gefunden ist oder die oberste Ebene des Programms erreicht ist. Die Ausnahme wird von der Hierarchieebene, wo sie aufgetreten ist, jeweils eine Hierarchieebene weiter nach oben gereicht.

Sofern eine Ausnahme nirgends aufgefangen wird (dies kann unter anderem bei Ausnahmen der Java-Standardklasse `Throwable` oder einer der direkten Subklassen unterbleiben, denn diese Ausnahmen müssen ja nicht explizit behandelt werden), bricht der Java-Interpreter normalerweise die Ausführung des Programms ab und erzeugt eine Bildschirmausgabe mit der entsprechenden Meldung, welche Ausnahme aufgetreten ist.



Bildschirmausgabe einer nichtbehandelten Standardausnahme.

Warnungen

Die `throws`-Anweisung darf nicht mit der `throw`-Sprunganweisung verwechselt werden (trotz einer gewissen Verwandtschaft).

Beispiele

Eine Methodendeklaration mit einer (selbstdeklarierten) Ausnahme:

```
public void eineMethode_mitExceptions() throws MeineException
{
  ...
}
```

Die Behandlung innerhalb einer `try-catch`-Struktur könnte so aussehen:

```
try
{
eineMethode_mitExceptions()
...
}
catch (MeineException e)
{
// Behandlung der Ausnahme.
// Das Ausnahmeobjekt e wird behandelt
...
}
```

Verwandte Befehle

```
break
catch
continue
return
throw
try
finally
```

transient

Ein Modifier zur Kennzeichnung von Variablen.

Beschreibung

Der vorangestellte Modifier `transient` gibt an, dass eine Variable nicht Teil eines persistenten Status eines Objektes ist.

Anwendung

Das Schlüsselwort `transient` war in den bisherigen Versionen von Java zwar geschützt, jedoch ohne konkrete Funktionalität, das bedeutet als »unused« markiert. Ab der Java-Version 2 (mit zugehörigem JDK 1.2) hat es eine definierte Bedeutung. Es ist sowohl als statische Variable in der Klasse `java.lang.reflect.Modifier` in Verwendung (genauer – dort repräsentiert `public static final int TRANSIENT` den `int`-Wert des `transient`-Modifiers), als auch direkt im Rahmen des CORBA-Konzeptes und im Bereich der verteilten Programmierung von Bedeutung.

Das Remote Method Invocation Interfaces (RMI) von Java für die verteilte Programmierung bietet die Möglichkeit, beliebige Java-Klassen, welche auf einer anderen virtuellen Maschine laufen, anzusprechen. Dabei ist es egal, ob die virtuelle Maschine lokal vorhanden oder irgendwo in einem Netzwerk ausgeführt wird. In Java wurden bereits in den 1.1.x-Varianten vier Pakete zur Umsetzung zur Verfügung gestellt (`java.rmi`, `java.rmi.dgc`, `java.rmi.registry` und `java.rmi.server`). In der Version 1.2 kam mit `java.rmi.activation` noch ein weiteres Paket hinzu. Die damit verbundene Erweiterung des RMI-Konzeptes 1.2 erlaubt es, Objekte anhand einer Referenz zu reaktivieren, wenn diese zuvor persistent gemacht wurden (Remote Object Activation). Für diesen Zweck stellt das JDK 1.2 die Programme `rmic` (Java RMI Stub Compiler), `rmiregistry` (Java Remote Object Registry) und – neu im JDK 1.2 – `rmid` (Java RMI Activation System Daemon) zur Verfügung.

In diesen Zusammenhang gehört das Konzept der sogenannten Object Serialization. Es ermöglicht das Abspeichern der Inhalte eines Objektes in einen Stream. Dieser kann zum Beispiel eine beliebige Datei sein. Ein Objekt kann so in einem Stream zwischengespeichert und zu einem späteren Zeitpunkt daraus wieder aufgebaut werden. Die Lebensdauer eines Objektes kann damit sogar über die eigentliche Laufzeit eines Programms hinaus verlängert werden. Genau genommen produziert die Object Serialization einen Stream mit Informationen über die Java-Klassen von gespeicherten Objekten. Für serialisierte Objekte werden dann in dem Stream genügend Informationen gespeichert, um diese Objekte sogar wiederherzustellen, wenn eine unterschiedliche (aber zumindest kompatible) Version von der Implementation der Klasse vorhanden ist. Das `Serializable`-Interface (`public interface Serializable` im Package `java.io`) ist definiert, um alle Klassen zu

identifizieren, welche das *serializable protocol* implementieren. Hauptanwendung für solche Serialisierungsprozesse ist das Versenden von Objekten über das Netzwerk im Zusammenhang mit dem RMI-Konzept.

Das Schlüsselwort `transient` erlaubt es nun, einzelne Variablen zu markieren, welche nicht Teil des persistenten Status eines solchen Objektes sein sollen. Also beispielsweise sensitive Informationen oder Variablen, welche für eine spätere Wiederherstellung von Objekten nichtrelevante Informationen enthalten. Direkte Sytemzugriffe, etwa Dateibehandlung, sind solche Informationen, welche nur relativ zu einem Adressraum Sinn machen und meist nicht in den persistenten Status eines Objektes geschrieben werden sollten. Um das sicherzustellen, werden Variablen mit solchen Informationen als `transient` deklariert. Sie werden damit nicht serialisiert.

Wenn eine Variable als `transient` deklariert ist, kann man dies beispielsweise mit der Methode `public boolean isTransient()` überprüfen, welche für den Fall den Wert `true` zurückgibt.

Um sensitive Informationen während des gesamten Prozesses der Serialisation zu schützen, kann man sie zusätzlich noch als `private` (also `private transient`) deklarieren.

Beispiele

Eine Klasse, in der nur die Variablen `x` und `y` gespeichert werden, wenn eine Instanz persistent gemacht wird – die Variable `z` wird nicht gespeichert.

```
class MeineKlasse
{
 int x, y;
 transient float z;
...
}
```

Verwandte Befehle

`synchronized`
`volatile`

true

Ein boolesches Literal.

Beschreibung

Das Schlüsselwort `true` beschreibt eine von zwei möglichen Ausprägungen von booleschen Variablen oder Ausdrücken. Es steht für richtig.

Anwendung

Java kennt nur zwei boolesche Literale: `true` und `false`. Diese sind die einzigen möglichen Ausprägungen von booleschen Variablen oder Ausdrücken.

Warnungen

Im Gegensatz zu vielen anderen Sprachen gibt es keine direkte Äquivalenz zwischen numerischen Werten ungleich 0 beziehungsweise gleich 0 und den beiden booleschen Literalen.

Beispiele

Eine boolesche Variablendeklaration:

```
boolean test=true;
```

Verwandte Befehle

```
boolean
false
```

try

Der zentrale Teil der Schutzanweisung zum Auffangen von Ausnahmen in Java.

Beschreibung

Java verfolgt zum Abfangen von Laufzeitfehlern ein Konzept, welches mit sogenannten Ausnahmen arbeitet. Ausnahmen werden unter Java über ein `try-catch-finally`-Konstrukt aufgefangen. Der `finally`-Zweig ist optional. Dabei sollten kritische Aktionen in einem Java-Programm immer innerhalb eines `try`-Blocks durchgeführt werden. Der Begriff »try« sagt bereits sehr treffend, was dort passiert. Es wird versucht, den Code innerhalb des `try`-Blocks auszuführen. Wenn dort eine Ausnahme ausgeworfen wird, wird dieses sofort entsprechend im zugehörigen `catch`-Block gehandhabt. Das Schlüsselwort macht nur Sinn in Zusammenhang mit mindestens einem nachfolgenden `catch`-Block. Am Ende eines `try`-Blocks können beliebig viele dieser `catch`-Klauseln stehen. Sie werden einfach nacheinander notiert.

Anwendung

Während der Laufzeit eines Programms kann es nichtplanbare Situationen geben, welche zur Laufzeit eines Programms auftreten und eine unmittelbare Reaktion durch das Programm respektive den Anwender notwendig machen. Etwa, wenn ein Zugriff auf ein Diskettenlaufwerk erfolgt, wo keine Diskette eingelegt ist, oder die Division durch Null während einer mathematischen Operation. Ein solcher Vorgang wird in Java eine Ausnahme erzeugen, welche dann unmittelbar vom Programm bearbeitet werden muss.

Damit diese Ausnahmen sicher gehandhabt werden können, stellt Java Schutzanweisungen zur Verfügung. Diese Anweisungen benutzen im Wesentlichen die drei Schlüsselworte `try`, `catch` und `finally` in Kombination.

Innerhalb des mit `try` eingeleiteten Blocks werden die Anweisungen notiert, welche eine Ausnahme erzeugen können. Kritische Aktionen in einem Java-Programm sollten immer innerhalb eines `try`-Blocks durchgeführt werden.

Innerhalb des mit `catch` eingeleiteten Blocks werden die Anweisungen notiert, welche beim Auftreten einer bestimmten Ausnahme durchgeführt werden sollen. Wenn also eine der Anweisungen innerhalb des `try`-Blocks ein Problem erzeugt, wird dieses durch die passende `catch`-Anweisung aufgefangen und entsprechend behandelt (sofern die `catch`-Anweisung dafür die passende Behandlung enthält – `catch`-Anweisungen haben Ausnahmeobjekte als Parameter). Am Ende eines `try`-Blocks können beliebig viele `catch`-Klauseln stehen. Sie werden einfach nacheinander notiert. Damit können unterschiedliche Arten von Ausnahmen auch verschiedenartig – und damit sehr qualifiziert – gehandhabt werden.

Die optionale `finally`-Anweisung erlaubt die Abwicklung wichtiger Abläufe (wie beispielsweise das Schließen von Dateien), bevor die Ausführung des gesamten `try-catch-finally`-Konstruktes unterbrochen wird. Der in dem `finally`-Block untergebrachte Code wird in jedem Fall abgearbeitet (ohne oder mit Auftreten einer Ausnahme).

Beispiele

Ein einfaches `try-catch`-Konstrukt mit einer `catch`-Klausel:

```
try
{
//  kritische Anwendungen
}
catch (Exception e)
{
// Behandlung der Ausnahme
// Das Ausnahmeobjekt e wird behandelt
}
```

Ein einfaches `try-catch`-Konstrukt mit einer `catch`- und einer `finally`-Klausel:

```
try
{
//  kritische Anwendungen
}
catch (Exception e)
{
// Behandlung der Ausnahme
// Das Ausnahmeobjekt e wird behandelt
}
finally
{
// Abschlußarbeiten
}
```

Ein `try-catch`-Konstrukt mit mehreren `catch`-Klauseln:

```
try
{
//  kritische Anwendungen
}
catch (Exception e)
{
// Behandlung der Ausnahme 1
// Das Ausnahmeobjekt e wird behandelt
}
catch (MeineException1 e)
{
// Behandlung der Ausnahme 2 - selbstdefiniert
}
```

```
catch (MeineException2 e)
{
// Behandlung der Ausnahme 3 - selbstdefiniert
}
```

Verwandte Befehle

```
catch
finally
switch
break
default
if
if-else
```

void

Ein leerer Datentyp zur Deklaration des Rückgabetyps einer Methode.

Beschreibung

Das Schlüsselwort `void` legt fest, dass eine Methode keinen Wert zurückgibt.

Anwendung

Rückgabewerte von Methoden sind zum einen wichtige Rückmeldungen einer Methode, ob sie korrekt abgearbeitet wurde oder was sie genau ausgeführt hat. Zum anderen können Rückgabewerte von Java-Methoden gezielt Resultate liefern. Rückgabewerte können von jedem erlaubten Datentyp sein, insbesondere auch Objekte (beispielsweise Zeichenketten).

Eine Methode muss immer einen Wert zurückgeben (und zwar genau den Datentyp, welcher in der Deklaration angegeben wurde), es sei denn, sie ist mit dem Schlüsselwort `void` deklariert worden, was gerade bedeutet, dass sie explizit keinen Rückgabewert hat.

Warnungen

Konstruktoren liefern nie einen Rückgabewert. Sie dürfen jedoch dennoch nicht als `void` deklariert werden.

Beispiele

Eine Methodendeklaration mit einem Rückgabewert vom Typ `void`:

```
void test();
```

Verwandte Befehle

```
boolean
byte
char
double
float
int
long
short
```

volatile

Eine Synchronisationsanweisung im Rahmen des Multithreadingkonzeptes.

Beschreibung

Das Schlüsselwort `volatile` erlaubt es, Variablen im Rahmen von Multithreading-Prozessen zu synchronisieren.

Anwendung

Java erlaubt Threads, im Rahmen von gemeinsam genutzten Variablen private Arbeitskopien von diesen Variablen zu erstellen. Damit soll eine möglichst effiziente Implementation von multiplen Threads ermöglicht werden. Java stellt nun mehrere Möglichkeiten zur Verfügung, das gesamte Management dieser Prozesse zu gewährleisten (Verbindung zwischen Originalvariable (Masterkopie) und Kopien, Synchronisation, gesperrt oder veränderbar usw.).

Eine Möglichkeit, um unter gewissen Umständen so ein Management durchzuführen, ist die Deklaration einer solchen Variablen mit dem Schlüsselwort `volatile`. Sofern eine Variable so deklariert ist, muss ein Thread seine Arbeitskopie der Variablen mit der Masterkopie bei jedem Zugriff auf die Variable abstimmen.

Das Schlüsselwort `volatile` ist eng verwandt mit `synchronized`, welches aber nicht direkt auf Variablen angewandt werden kann. Der Mechanismus des Deklarierens einer Variablen als `volatile` überträgt und erweitert jedoch den Synchronisationsmechanismus auch auf Variablen. Insbesondere gibt es Fälle, wo eine Synchronisation von Methoden nicht unbedingt sinnvoll ist, wenn nur außerhalb deklarierte und in den synchronisierten Methoden enthaltene Variablen korrekt upgedatet werden sollen, bevor der Kontrollfluss von einer Methode (einem Thread) zu einer weiteren Methode (Thread) weitergegeben wird. So etwas kann auch über reine Deklaration der Variablen als `volatile` erreicht werden.

Warnungen

Eine als `final` deklarierte Variable darf nicht zudem als `volatile` deklariert werden. Ein Compiler-Fehler wird die Folge sein.

Beispiele

Eine als `volatile` deklarierte Variable:

```
static volatile int i = 42;
```

Verwandte Befehle

`synchronized`
`transient`

while

Eine der drei Iterationsanweisungen von Java.

Beschreibung

Die `while`-Anweisung gibt an, unter welchen Bedingungen eine Schleife gestartet und abgebrochen werden soll.

Anwendung

Das Schlüsselwort `while` wird in Java in zwei Fällen verwendet:

- als eigenständige `while`-Anweisung
- als Ende der `do`-Anweisung (die deshalb statt `do`-Anweisung auch oft `do-while`-Anweisung genannt wird)

Die `while`-Anweisung (wie auch die `do-while`-Anweisung) testet eine boolesche Variable oder einen Ausdruck. Solange der Test den Wert `true` liefert, wird die nachfolgende Unteranweisung oder der nachfolgende Block ausgeführt. Erst wenn die boolesche Variable oder der Ausdruck den Wert `false` ausweist, wird die Wiederholung eingestellt und die Schleife verlassen. Es gilt folgende Syntax:

```
while(Bedingung)
[Unteranweisung oder Block]
```

Die `do-while`-Anweisung hat folgende Syntax:

```
do
[Unteranweisung oder Block]
while(Bedingung)
```

Warnungen

- Der Codeblock innerhalb der `while`-Anweisung wird nur dann ausgeführt, wenn die überprüfte Bedingung erfüllt ist. Rein von der Syntax her kann man es sich dadurch verdeutlichen, dass die Überprüfung am Anfang der Struktur steht und deshalb die Überprüfung dort stattfindet. Dies ist ein ganz wichtiger Unterschied zur verwandten `do-while`-Schleife, die auf jeden Fall mindestens einmal ausgeführt wird, egal ob die Bedingung erfüllt ist oder nicht.
- Die Syntax

  ```
  while(true)
  [Unteranweisung oder Block]
  ```

  ist syntaktisch vollkommen korrekt. Dadurch wird eine Endlosschleife erzeugt. Ebenso entsteht eine Endlosschleife, wenn eine als Testkriterium überprüfte Variable innerhalb der Schleife überhaupt nicht verändert oder so verändert wird, dass das Abbruchkriterium nie erreicht wird.

Beispiele

Eine einfache `while`-Schleife:

```
while (testvariable == true)
  {
  ...
  }
```

Tips

Die Erzeugung einer Endlosschleife kann ein Fehler sein, aber auch sinnvoll verwendet werden, wenn im Rahmen von Multithreading darin ein Thread permanent laufen soll.

Verwandte Befehle

`do`
`for`

Die Pakete der Java-2-Plattform

Java ist in Form von Paketen aufgebaut. Diese bilden das Java-API, welches von Programmierern zur Erstellung ihrer Anwendungen genutzt werden kann. Wesentlicher Vorteil ist, dass nicht jedes Mal »das Rad neu erfunden« werden muss, sondern einfach vorhandene Elemente direkt benutzt oder Erweiterungen von bestehenden Elementen vorgenommen werden. In diesem Kapitel soll ein allgemeiner Überblick über das Standard-API vom JDK-1.2-Final erfolgen. Dies bedeutet, die darin enthaltenen Pakete sind auf jeden Fall in der aktuellen Java-Implementierung vorhanden. Dieser Abschnitt ist auch deshalb interessant, weil sich nur so die ganzen 1.1- und 1.2-Erweiterungen / Veränderungen / Verlagerungen von Java nachvollziehen lassen. Das Java-API ist mittlerweile jedoch so umfangreich geworden, dass wir jedoch eine Auswahl dessen treffen müssen, was hier dargestellt werden kann.

Das Java-1.2-API wurde erheblich gegenüber dem 1.1-API erweitert. Es hat sich aber nicht nur zwischen der Version 1.1 mit den diversen Zwischenversionen und der Version 1.2 verändert. Gravierende Modifikationen fanden über die einzelnen Betaversionen des 1.2-API statt. Das API wurde in der Struktur teilweise vollkommen umsortiert. Wir stellen daher zuerst das 1.2-API der Beta 2/3[7] dem Final direkt gegenüber.

Die nachfolgende Tabelle stellt die einzelnen Pakete des Java-1.2-Standard-API (also der Java-2-Plattform) im direkten Vergleich zu der bisherigen Struktur dar. Dabei ist die bisherige Paketstruktur auf der linken Seite der Tabelle der neuen Struktur (rechte Seite) gegenübergestellt. Die Zuordnungsnummer in der ersten Spalte soll die Orientierung erleichtern, wenn Sie ein Paket der einen Version in der jeweils anderen Struktur suchen. »*« steht dafür, dass eine eindeutige Zuordnung zu einem Paket in der anderen Struktur nicht eindeutig möglich ist. Entweder ist die Funktionalität dort nicht vorhanden oder es ist auf andere Pakete aufgeteilt.

Nr	1.2-API Beta	1.2-API Final
1	`Java.applet`	`java.applet`
2	`Java.awt`	`java.awt`
3	`Java.awt.accessibility`	*52*
4	`Java.awt.color`	`java.awt.color`

7. **Da die Betaversionen des JDK wie auch das Final kostenlos und leicht aus dem Internet zu laden waren, und vor allem das Final mehrere Monate länger als angekündigt auf sich warten ließ, haben sich zahlreiche Entwickler (obgleich nur Betaversionen) damit beschäftigt. Aber die Gegenüberstellung betrifft natürlich auch die 1.1.x-Versionen, denn die Betaversionen 2/3 des JKD 1.2 repräsentieren im wesentlichen das Ende der 1.1.x-Strukturen.**

5	Java.awt.datatransfer	java.awt.datatransfer
6	Java.awt.dnd	java.awt.dnd
7	Java.awt.event	java.awt.event
8	Java.awt.font	java.awt.font
9	Java.awt.geom	java.awt.geom
10	Java.awt.im	java.awt.im
11	Java.awt.image	java.awt.image
12	*	java.awt.image.renderable
13	Java.awt.print	java.awt.print
14	Java.awt.swing	53
15	Java.awt.swing.basic	59
16	java.awt.swing.beaninfo	*
17	java.awt.swing.border	54
18	java.awt.swing.event	56
19	java.awt.swing.jlf	*
20	java.awt.swing.motif	*
21	java.awt.swing.multi	61
22	java.awt.swing.plaf	58
23	java.awt.swing.table	62
24	java.awt.swing.target	*
25	java.awt.swing.text	63
26	java.awt.swing.tree	67
27	java.awt.swing.undo	68
28	java.beans	java.beans
29	java.beans.beancontext	java.beans.beancontext
30	java.io	java.io
31	java.lang	java.lang
32	java.lang.ref	java.lang.ref
33	java.lang.reflect	java.lang.reflect
34	java.math	java.math
35	java.net	java.net
36	java.rmi	java.rmi
37	java.rmi.activation	java.rmi.activation
38	java.rmi.dgc	java.rmi.dgc
39	java.rmi.registry	java.rmi.registry
40	java.rmi.server	java.rmi.server
41	java.security	java.security

42	java.security.acl	java.security.acl
43	java.security.cert	java.security.cert
44	java.security.interface	java.security.interfaces
45	java.security.spec	java.security.spec
46	java.sql	java.sql
47	java.text	java.text
48	java.util	java.util
49	java.util.jar	java.util.jar
50	java.util.mime	*
51	java.util.zip	java.util.zip
52	3	javax.accessibility
53	14	javax.swing
54	17	javax.swing.border
55	*	javax.swing.colorchooser
56	18	javax.swing.event
57	*	javax.swing.filechooser
58	22	javax.swing.plaf
59	*	javax.swing.plaf.basic
60	*	javax.swing.plaf.metal
61	21	javax.swing.plaf.multi
62	23	javax.swing.table
63	25	javax.swing.text
64	*	javax.swing.text.html
65	*	javax.swing.text.html.parser
66	*	javax.swing.text.rtf
67	26	javax.swing.tree
68	27	javax.swing.undo
69	org.omg.CORBA	org.omg.CORBA
70	*	org.omg.CORBA.DynAnyPackage
71	org.omg.CORBA.ContainedPackage	*
72	org.omg.CORBA.ContainerPackage	*
73	org.omg.CORBA.InterfaceDefPackage	*
74	org.omg.CORBA.ORBPackage	org.omg.CORBA.ORBPackage
75	org.omg.CORBA.portable	org.omg.CORBA.portable
76	org.omg.CORBA.TypeCodePackage	org.omg.CORBA.TypeCodePackage
77	org.omg.CosNaming	org.omg.CosNaming
78	org.omg.CosNaming. NamingContextPackage	org.omg.CosNaming. NamingContextPackage

Die neue Java-2.0-Plattform beinhaltet also einen großen Komplex von Paketen. Nach Aussage von Sun sei damit der Funktionsumfang der Kernplattform weitgehend komplett und zukünftig soll Entwicklern eine permanente Umstrukturierung / Erweiterung wie über die bisherigen (Zwischen-) Versionen erspart bleiben. Von den vielen Paketen bilden fünfzehn Basispakete (die sogenannten Basisklassen) das Rückgrat der Java-2.0-Technologie:

```
applet
awt
beans
io
lang
math
net
rmi
security
sql
text
util
accessibility
swing
corba
```

Die Details

Gehört die Behandlung des Java-API in ein Referenzbuch über Java? Die Antwort mag vorschnell lauten, dass dies überhaupt keine zulässige Frage ist. Das API der Java-2-Plattform gehört zwingend in ein Buch über den aktuellen Java-Stand aufgenommen. Aber warum eigentlich? Wenn der (leicht hinkende) Vergleich zu einer Referenz einer menschlichen Sprache wie Deutsch betrachtet wird, würde die Frage dort lauten, ob die gesammelten Werke von Goethe und Schiller in ein Referenzbuch der deutschen Sprache gehört. Dort lautet die Antwort wahrscheinlich, dass in ein Referenzbuch nur die Worte und die Grammatik der Sprache gehören. Höchstens als Beispiele Zitate von berühmten Sprachgelehrten.

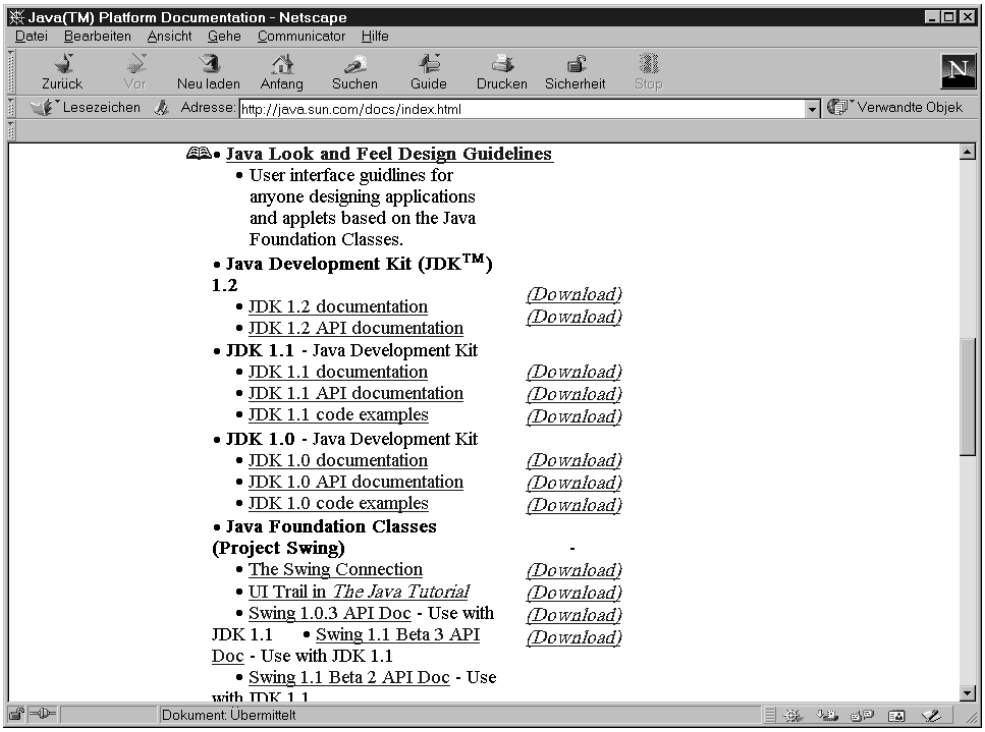
So wenig wie die Werke von Goethe und Schiller Grundlagen der deutschen Sprache sind (sie sind Resultate, was man mit der Sprache anstellen kann), so wenig gehört das API im engen Sinn zur Sprache Java. Das API ist ein Resultat dessen, was die Sprache Java leisten kann.

Dennoch – eine Programmiersprache unterscheidet sich natürlich erheblich von einer normalen Sprache. Insbesondere lässt sich Java nicht nur als Sprache fassen, sondern versteht sich als eine vollständige Plattform, zu deren Umfang im engeren Sinn die Java-Standardpakete gehören. Dementsprechend finden Sie nachfolgend die Behandlung der Paketstruktur der Java-2-Plattform.

Es stellt sich aber immer noch die Frage, was von dem gigantischen Umfang der Details besprochen werden soll? Die riesige Menge an Informationen macht in jedem Fall eine Einschränkung notwendig. Wir werden uns bei der Behandlung des Java-API auf eine allgemeine Übersicht beschränken müssen, was bedeuten soll, dass für eine vollständige Information auf weiterführende Literatur und vor allem die 1.2-API-Dokumentation von der Java-Webseite von Sun (`http://java.sun.com/docs/index.html`) verwiesen sei (siehe Abbildung).

Sie finden in diesem, dem vorangehenden und dem folgenden Abschnitt im Rahmen des Kapitels

- die Paket-Hierarchie des Java-API 1.2 (Final)
- die Veränderungen der Paket-Struktur zwischen den bisherigen APIs (inklusive Beta 3) und dem 1.2-Final-API
- eine Beschreibung der Standardpakete des Java-API 1.2 (Final) mit wichtigen Details
- die JDK-Version, ab der das Paket in das API integriert wurde
- die Bestandteile des Paketes (enthaltene Klassen, Schnittstellen, Ausnahmen usw.)
- und die veralteten Elemente des Java-API 1.2 (Final) inklusive der Information, ab welchem JDK diese als deprecated gelten.

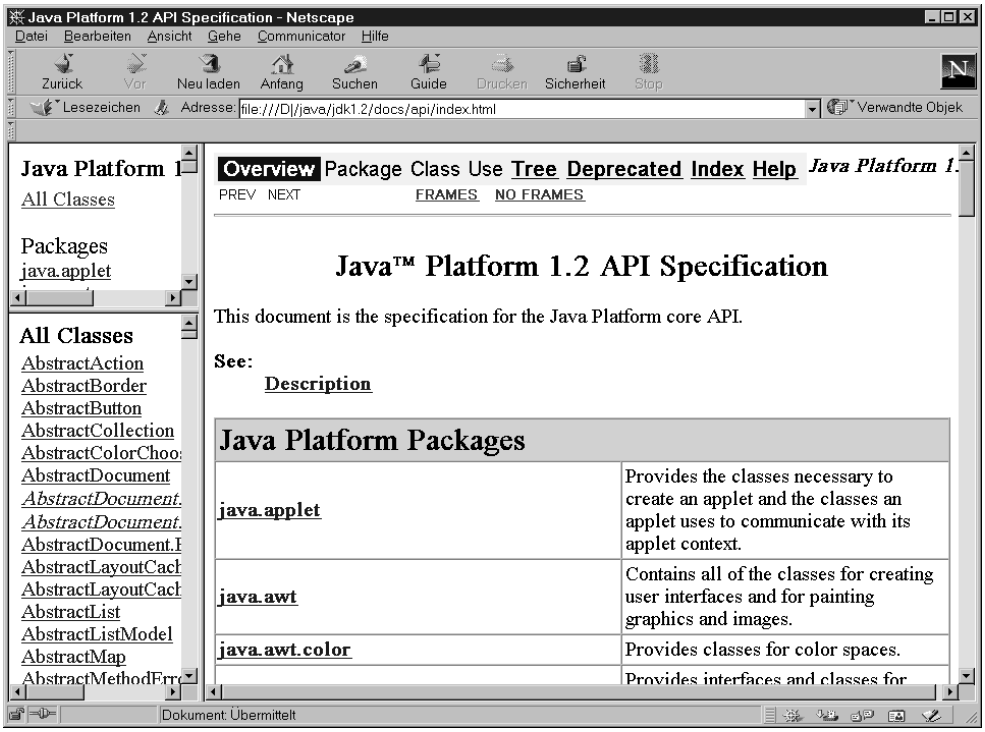


Hier gibt es die Dokumentation des JDK 1.2.

Diese Auswahl, was konkret in dem Kapitel behandelt wird, verfolgt ein etwas anderes Konzept, als es in einigen anderen Referenzen über Java zu finden ist und vielleicht auch hier erwartet wird. Insbesondere werden Sie bei einer ersten Betrachtung die vollständigen Angaben der Methoden und ihrer Parameter, sowie ihrer Bedeutungen und Informationen über Variablen in den Klassen vermissen. Damit fehlt auf den ersten Blick scheinbar einiges von dem, was man als Programmierer strenggenommen am dringendsten benötigt. Diese Vorgehensweise erfolgt aber nicht ohne Gründe. Im wesentlichen ist es das Motiv, dass das API bereits vollständig in der frei verfügbaren API-Online-Dokumentation erklärt wird[8].

Eine Angabe und Beschreibung der Methoden/Variablen würde in zentralen Bereichen auf eine Übersetzung und knappe Erläuterung dieser Dokumentation hinauslaufen. Der Mehrwert für den Leser im Verhältnis zum Aufwand[9] ist diskussions-

8. **Zwar auf Englisch, aber wer Java programmiert, wird meist auch mit Englisch einigermaßen klar kommen (hoffentlich ;-)).**
9. **Die Anzahl der Methoden und Variablen ist so gigantisch, daß - selbst falls nur eine Zeile Erklärung pro Methode/Variable erfolgen würde - 1000 Mehrseiten für eine vollständige Behandlung nur eine untere Schätzung abgeben. Selbst die Beschränkung auf die vollständige Erklärung sämtlicher Klassen, Schnittstellen, Ausnahmen und Errors würde bei nur einer Zeile mehrere hundert Mehrseiten bewirken.**



Die API-Online-Dokumentation im Browser

würdig. Böswillig ausgedrückt erhält er in diesem Fall nur das, was er bereits frei verfügbar sowieso zur Verfügung hat[10]. So ganz wollen wir aber natürlich nicht auf Paketdetails wie Variablen oder Methoden verzichten. Es wird jedoch nur eine Auswahl sein. In der hier gewählten Zusammenstellung der Informationen über das API erhält der Leser konkret folgenden Nutzwert:

- Eine deutsche Erklärung der wichtigsten Details eines Paketes.
- Die Angaben sämtlicher (!) Klassen, Schnittstellen, Errors und Ausnahmen des jeweiligen Paketes für die Java-2-Plattform in kompakter Form. Vor allem werden diese Angaben in einer Darstellung präsentiert, wie sie so in der frei verfügbaren API-Online-Dokumentation nicht zu finden ist[11]. In der API-Dokumentation findet der Leser zwar ebenso die Klassen, Schnittstellen, Errors und Ausnahmen als Auflistung bei den jeweiligen Paketen, aber immer ohne Angabe von Modifiern, Schnittstellen und Superklasse. Auf der anderen Seite sind diese Informationen über Modifiern, Schnittstellen und Superklasse zwar jeweils auf einer einzelnen Webseite in der Online-Dokumentation (jeweils ein

10. Obwohl es bei objektiver Betrachtung auch für eine solche Angabe sehr gute Argumente gibt – wir wollen jedoch einen anderen Weg gehen.
11. Es ist also nicht nur eine einfache »Erwähnung« der enthaltenen Klassen, Schnittstellen, Errors und Ausnahmen.

Hyperlink aus der Inhaltsangabe) zu finden, aber dort findet man keine Übersicht über andere Elemente des Pakets. In der hier gewählten Darstellung werden über die Erklärung des Paketes hinaus die Informationen über Klassen, Schnittstellen, Errors und Ausnahmen neu zusammengestellt und der Leser hat damit[12] folgende Mehrwerte gegenüber der API-Dokumentation:

  - Eine Übersicht über sämtliche Elemente (auf Klassenebene) eines Paketes auf einen Blick
  - Eine Übersicht über direkte Vererbungen
  - Eine Übersicht über implementierte Schnittstellen
- Eine Unterstützung, um mit einer bösen Java-Krankheit – den permanenten Veränderungen und den Inkompatibilitäten zwischen den zahlreichen Zwischen- und Betaversionen – sinnvoll umgehen zu können. Zum einen ist der Abschnitt über die als deprecated bezeichneten Elemente dabei eine zwingend notwendige Informationsquelle. Aber leider wurden auch zahlreiche Elemente in Java über die Zeit verändert oder weggelassen, ohne dass sie in diese offizielle Liste aufgenommen wurden. In diesem Fall kann aber unter Umständen ein Blick auf die aktuelle Paketdarstellung helfen oder die Pakethierarchie des Java-API 1.2 (Final), die Angabe der Veränderungen der Paketstruktur zwischen den bisherigen APIs (inklusive Beta 3) und dem 1.2-Final-API.

Um das Kapitel nun sinnvoll zu nutzen, sind unter anderem folgende Ansätze angedacht[13]:

- Sie wollen sich allgemein über den API-Aufbau informieren.
- Sie benötigen Informationen über direkte Vererbungen von Klassen, Schnittstellen, Ausnahmen oder Errors.
- Sie benötigen Aufschluß über implementierte Schnittstellen.
- Sie benötigen allgemeine Informationen über die bereitgestellten Funktionalitäten eines Pakets.
- Sie wollen sich über sämtliche enthaltenen Klassen, Schnittstellen, Errors und Ausnahmen eines Paketes kompakt informieren.
- Sie benötigen Informationen über Veränderungen in dem gesamten API.
- Bei Kenntnis eines Klassen-, Ausnahmen-, Error- oder Schnittstellennamens suchen Sie im Index nach der konkreten Seite, wo sie in diesem Abschnitt behandelt wird. Falls Ihnen die weiteren Informationen (Paket, Vererbung, Schnittstellen, Modifier, etc.) in diesem Abschnitt dann noch nicht genügen, erleichtern sie zumindest ein schnelles Auffinden in der API-Online-Dokumentation. Dort finden sich dann weitergehende Angaben – etwa zu konkreten Methoden/Variablen etc.

12. zumindest soll es so sein ;-)

13. Dabei ist die parallele Verwendung der offiziellen - und frei verfügbaren - API-Online-Dokumentation bei Bedarf ausdrücklich empfohlen.

Schauen wir uns nun die einzelnen Pakete an. Dabei verfolgen wir bei jedem besprochenen Paket folgenden Aufbau:

Kategorie	Was steht da	Wann vorhanden
Titelzeile	Der Name des Paketes, welches beschrieben werden soll.	Immer vorhanden
Beschreibung	Eine Kurzbeschreibung der Bedeutung des Paketes.	Immer vorhanden
Anwendung	Die eigentliche Referenz. Hier erfolgt die Beschreibung der Verwendung des Paketes und wichtiger Details (darunter fallen in besonders wichtigen Fällen auch die Beschreibungen von Methoden und Variablen).	Immer vorhanden
JDK-Version	Seit welcher JDK-Version ist das Paket vorhanden (nicht einzelne Elemente, die auch später hinzugekommen sein können).	Immer vorhanden
Warnungen	Eventuelle Warnungen im Umgang mit dem Paket	Optional
Klassen	Enthaltene Klassen immer mit vollständiger Unterschrift, also , inklusive Modifier, optionaler Superklasse und optional implementierenden Schnittstellen.	Immer vorhanden
Schnittstellen	Enthaltene Schnittstellen immer mit vollständiger Unterschrift, also inklusive Modifier, optionaler Superschnittstelle und optional implementierenden Schnittstellen.	Immer vorhanden
Ausnahmen	Enthaltene Ausnahmen immer mit vollständiger Unterschrift, also inklusive Modifier, optionaler Superklasse und optional implementierenden Schnittstellen.	Immer vorhanden
Errors	Enthaltene Error-Klassen immer mit vollständiger Unterschrift, also inklusive Modifier, optionaler Superklasse und optional implementierenden Schnittstellen.	Optional
Verwandte Pakete	Sofern andere Pakete in einem relevanten Sinnzusammenhang zu dem besprochenen Paket stehen, werden sie hier angegeben.	Optional

Hinweis: Java stellt mit seinem Paketaufbau einen Sonderfall[14] in der Welt der Programmiersprache dar. Bei der Besprechung der Paketstruktur von Java müssen wir deshalb teilweise etwas von dem bisher und in verwandten Titeln der Buchreihe durchgängig verwendeten Aufbau abweichen.
Die grundsätzliche Angabe aller optionalen Modifier etc. einer Klasse / Schnittstelle / Ausnahme / Fehlerklasse erlaubt neben dem meist aussagekräftigen Namen bereits eine ziemlich genaue Einschätzung der Funktion und der konkreten Leistung.

14. Wenngleich keinen Einzelfall

java.applet

Der Ursprung aller Applets.

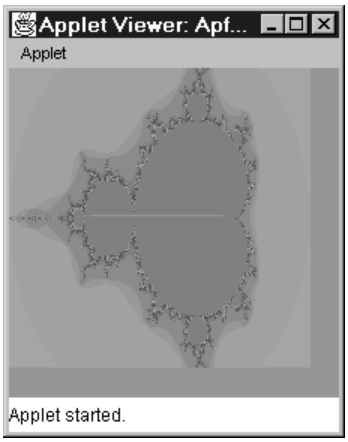
Beschreibung

Das Paket ist die Grundlage von allen Java-Applets. Insbesondere muss jedes Applet als eine Subklasse der darin enthaltenen Klasse `Applet` erzeugt werden. Diese Klasse beinhaltet bereits sämtliche Eigenschaften, welche die Kooperation mit dem Container sicherstellen, die Fähigkeiten des AWT (Abstract Windowing Toolkit) für Oberflächenprogrammierung (Menüs, Button, Mausereignisse,...) nutzbar machen, und die lebenswichtigen Grundmethoden für ein Applet. Es gibt in der Klasse zwar keine Variablen, aber diese Klasse stellt mit den Methoden

```
public void destroy(), public AppletContext getAppletContext(), public String
getAppletInfo(), public AudioClip getAudioClip(URL url), public AudioClip
getAudioClip(URL url, String name), public URL getCodeBase(), public URL
getDocumentBase(), public Image getImage(URL url), public Image getImage(URL
url, String name), public Locale getLocale(), public String getParameter(String
name), public String[][] getParameterInfo(), public void init(), public boolean
isActive(), public static final AudioClip newAudioClip(URL url), public void
play(URL url), public void play(URL url, String name), public void
resize(Dimension d), public void resize(int width, int height), public final
void setStub(AppletStub stub), public void showStatus(String msg), public void
start() und public void stop()
```

sowie der Subklasse `public class JApplet extends Applet implements Accessible, RootPaneContainer`

alle Möglichkeiten bereit, die Funktionalität eines Applets zu gewährleisten.



Ein Applet im Appletviewer des JDK.

Anwendung

Wichtigste Anwendung des Paketes ist die Bereitstellung der `Applet`-Klasse – die (!) zentrale Applet-Klasse von Java. Diese Klasse ist die Superklasse von allen Applets und muss von allen Applets implementiert werden. Jedes Applet wird mit folgender Syntax erstellt:

```
public class [AppletName] extends java.applet.Applet
```

Java setzt zwingend voraus, dass eine Applet-Klasse als `public` deklariert wird. Damit muss auch die Datei mit dem Quelltext zwingend unter dem Namen diese Klasse (mit der Erweiterung `.java`) abgespeichert werden. Aus dieser Datei erzeugt der Compiler dann eine `.class`-Datei mit gleichen Namen, die dann in einer HTML-Datei referenziert wird (etwa mit dem `Applet`-Tag). Zum Aufruf eines Applets innerhalb einer Webseite genügt eine einfache HTML-Struktur der folgenden Form:

```
<html>
<body>
<applet code=[.class-Datei des Applets] width=[numerische Breitenangabe]
height=[numerische Höhenangabe]>
</applet>
</body>
</html>
```

Das Applet kann dabei mit oder ohne die Erweiterung angegeben werden. Optional ist bei der Referenzierung noch die Angabe des Namens der Java-Datei (der Parameter `name`).

Java-Applets unterscheiden sich essenziell von eigenständig lauffähigen Java-Applikationen. Eine solche Applikation benötigt immer eine `main()`-Methode, die als erste Methode beim Laden des Programms gestartet wird und mit deren Beendigung auch das Programm endet. Ein Java-Applet hingegen besitzt im Gegensatz dazu überhaupt keine `main()`-Methode[15]. Statt dessen werden einem Applet über seine Superklasse eine Anzahl von Methoden bereitgestellt, die dessen Grundfunktionen sicherstellen. Bei voller Funktionsfähigkeit handelt es sich um mindestens vier Methoden:

- die `init()`-Methode
- die `start()`-Methode
- die `stop()`-Methode
- die `destroy()`-Methode

15. **Es gibt jedoch die Möglichkeit, eine Applikation so zu erstellen, dass sie sowohl als Applet, als auch als eigenständiges Programm fungieren kann. Dann wird sich dort auch eine solche Methode finden.**

Diese Grundmethoden sorgen dafür, dass sich ein Applet in seiner Umgebung korrekt verhält, und können in jedem Applet überschrieben werden, um sie entsprechend der Funktionalität des Applets anzupassen.

Die Methode `public void init()` bedeutet dabei den Einstieg in ein Applet-Leben in Form einer Initialisierung. Die Methode wird automatisch aufgerufen, sobald ein Applet in den Browser (oder einen anderen Container wie den Appletviewer) das erste Mal geladen wird. Hier wird das System zur Ausführung des Applets vorbereitet. An dieser Stelle wird beispielsweise die Bildschirmanzeige initialisiert, die Größe eines Applets festgelegt (mit der Methode `resize()`) oder Threads eingerichtet. In der `init()`-Methode ist auch die Stelle zu finden, wo Parameter aus dem HTML-Code der Seite eingelesen werden (beispielsweise mit der Methode `public String getParameter(String name)`).

Die Methode `public void start()` startet das eigentliche Applet direkt nach der Initialisierung. Während die Initialisierung nur einmal erfolgt, kann (und wird normalerweise) der Aufruf der Startmethode beliebig oft wiederholt werden. Beispielsweise wenn das Applet gestoppt wurde oder das Applet durch Scrollen des HTML-Dokumentes erneut in den sichtbaren Bereich des Browsers geholt wird. Wenn die `start()`-Methode in einem konkreten Applet überschrieben wird, können dort diverse Funktionalitäten stehen. Beispielsweise das Starten eines Threads oder der Aufruf einer anderen Methode.

Das Ende der Aktivität eines Applets ist der Aufruf der Methode `public void stop()`. Sie wird beispielsweise automatisch aufgerufen, sobald die HTML-Seite, auf der das Applet eingebunden ist, verlassen wird. Ein gestopptes Applet ist jedoch noch weiter voll funktionsfähig und wird bei der Rückkehr auf die Seite automatisch durch die Funktion `start()` wieder zu neuem Leben erweckt (ohne erneute Initialisierung).

Wenn ein Applet-Container beendet wird, ermöglicht die Methode `public void destroy()` dem Applet, Aufräumarbeiten durchzuführen und belegte Systemressourcen freizugeben. Unter solche Aufräumarbeiten fallen beispielsweise die Beendigung von laufenden Threads oder die Freigabe von anderen laufenden Objekten. Normalerweise muss die `destroy()`-Methode jedoch nicht explizit überschrieben werden.

Neben den Grundmethoden stellt die Applet-Klasse noch eine Vielzahl weiterer Methoden bereit.

Die Methode `public final void setStub(AppletStub stub)` wird beispielsweise verwendet, um eine neue `AppletStub`-Schnittstelle (ebenfalls ein Bestandteil dieses Paketes) zu erstellen. Normalerweise wird der `AppletStub` automatisch durch das System gesetzt. `public boolean isActive()` zeigt hingegen an, ob ein Applet gerade aktiv ist oder gestoppt wurde. Der Rückgabewert ist `true`, wenn das Applet aktiv ist, sonst `false`.

Ganz wichtig für viele Applet-Aktivitäten sind die Methoden `public URL getDocumentBase()` und `public URL getCodeBase()`, welche den Dokumenten-URL respektive den URL des aktuellen Verzeichnisses (die aktuelle Pfadangabe des Applets) zurückgeben.

Um Übergabeparameter aus der Referenzierung in der HTML-Datei in einem Applet auswerten zu können, steht die Methode `public String getParameter(String name)` zur Verfügung. Als Parameter muss genau der Name spezifiziert werden, der im Parameter-Tag des Applets in den Parameterangaben als `NAME` angegeben wurde. Auch für Applets gilt (wie bei eigenständigen Applikationen), dass Parameter als Zeichenketten eingelesen werden. Eventuell in anderem Format benötigte Übergabeparameter müssen konvertiert werden, bevor sie in einem anderen Format verwendet werden können. Die Angabe von Parametern für das Applet unter HTML erfolgt einfach über Parameter-Tags – gegebenenfalls mehrere hintereinander. In diesem Fall spezifiziert jedes Parameter-Tag einen anderen Applet-Parameter.

Sie sehen wie folgt aus:

```
<PARAM name=[name] value=[wert]>
```

In den Parameterangaben werden zwei Bezeichner verwendet:

- `name` – der Name des Parameters
- `value` – der Wert des angegebenen Parameters

Eine Einbindung eines Java-Applets mit Parameterangaben in einer Webseite könnte so aussehen:

```
<APPLET code="MeinApplet.class" width=200 height=300>
  <PARAM name=Vorname value="Hans">
  <PARAM name=Nachname value="Dampf">
</APPLET>
```

Das gesamte `<PARAM>`-Tag unterscheidet, außer bei dem `value`-Wert, nicht zwischen Groß- und Kleinschreibung. Die Angaben sind ja ausschließlich unter HTML relevant, ob Groß- und Kleinschreibung nicht unterschieden wird. Ob der `value`-Wert Groß- und Kleinschreibung unterscheidet, hängt nur von dem Java-Code ab, der ihn verwendet.

Wenn man Informationen über die Umgebung eines Applets erhalten will, kann man `public AppletContext getAppletContext()` nutzen. Diese Methode gibt ein Objekt `AppletContext` (eine Schnittstelle) zurück, das dazu verwendet werden kann, solche Informationen, aber auch die Kontrolle über die Umgebung eines Applets zu erlangen. Die Methoden in der Schnittstelle `AppletContext` wiederum ermöglichen es herauszufinden, welche Applets außer dem spezifizierten noch auf derselben Seite laufen, den Kontext eines anderen Applets zu importieren und dessen Methoden laufen zu lassen, Meldungen an andere Applets weiterzugeben oder Bild- und Ton-Clips zu importieren.

Konkret enthält die Schnittstelle `AppletContext` unter anderem Methoden wie `public AudioClip getAudioClip(URL url)` zum Kreieren eines Audioclips, `public Image getImage(URL url)` zur Rückgabe von einem `Image`-Objekt, `public void showDocument(URL url)` beziehungsweise `public void showDocument(URL url, String target)` zum Ersetzen der gerade angezeigten Webseite durch den angegebenen URL (dabei bedeutet `url` einen absoluten URL für das Dokument und `target` einen String zur Angabe, wo die Seite angezeigt werden soll[16]).

Eine weitere Methode der Klasse `Applet` selbst – `public String getAppletInfo()` – gibt die Informationen über ein Applet als String zurück, die dort zur Verfügung gestellt werden (beispielsweise den Applet-Namen, die Version, Datum, Autor, Copyright mit Datum u.ä.).

`public Locale getLocale()` gibt – sofern gesetzt – den Ort des Applets zurück. Die Methode `public String[][] getParameterInfo()` liefert ein Zeichenketten-Array, das alle Parameter auflistet, welche das Applet interpretieren kann. Diese Zeichenkette spezifiziert für jeden Parameter den Namen, den Typ und eine Beschreibung. In den Zusammenhang fällt auch die gezielte Ausgabe von Informationen durch das Applet über `public void showStatus(String msg)`, mit der ein Java-Applet eine Meldung (den Parameter) im Statusbereich des Browsers anzeigen kann.

Wenn ein Applet innerhalb des Applet-Codes in der Größe verändert werden soll (eine voreingestellte Größe hat es ja immer über die Größenangaben in der aufrufenden HTML-Datei), stehen mit `public void appletResize(int width, int height)` – `width` ist die Breite und `height` die Höhe – oder `public void resize(Dimension dim)` – `dim` ist ein Objekt mit der neuen Breite und Höhe – zwei Methoden bereit, die das bewerkstelligen.

Auch für den weiten Bereich Multimedia bietet die Applet-Klasse mannigfaltige Unterstützung. Die Methode `public Image getImage(URL url, String name)` beispielsweise lädt über den angegebenen absoluten URL Bild-Dateien das über `name` spezifizierte Bild (relative Angabe zu dem URL-Argument) in ein Applet, `public static final AudioClip newAudioClip(URL url)`, `public AudioClip getAudioClip(URL url)` oder `public AudioClip getAudioClip(URL url, String name)` hingegen sogar einen vollständigen Audioclip aus dem angegebenen URL. Für die Wiedergabe von Audiodateien gibt es `public void play(URL url)` oder `public void play(URL url, String name)`.

Die weitere Unterstützung von Multimedia innerhalb eines Applets ist im Wesentlichen in der Schnittstelle `public interface AudioClip` realisiert. Das `AudioClip`-Interface ist eine einfache Abstraktion zum Abspielen von einem Soundclip. Ab der Javaversion 1.2 können MIDI-Dateien (Typ 0 und Typ 1) sowie RMF-, WAVE-,

16. **Dabei sind für `target` die Angaben `_self` (Anzeige in dem Fenster oder Frame, wo sich das Applet befindet), `_parent` (Anzeige in dem Parentframe – wenn es kein übergeordnetes Parentframe gibt, ist die Angabe identisch mit `_self`), `_top` (Anzeige in dem Toplevel-Frame von dem Appletfenster), `_blank` (Anzeige in einem neuen Toplevel-Fenster ohne Namen) und die direkte Angabe des Namens eines Fensters oder Frames erlaubt.**

AIFF-, und AU-Dateien in hoher Tonqualität abgespielt und zusammengemischt werden. Wichtige Methoden dort sind `public void play()` zum Laden und direkten Abspielen von einem Sound-Clip und `public void loop()` zum Abspielen einer geladenen Tondatei in einer Schleife. `public void stop()` hält einen Multimedia-Clip an. Daneben erlauben die anderen in dem Paket enthaltenen Schnittstellen eine Erweiterung der Applet-Fähigkeiten in Bezug auf seine Umgebung, sowie Multimedia-Fähigkeiten. Die Schnittstelle `public abstract interface AppletContext` korrespondiert mit einer Applet-Umgebung, das heisst dem Dokument, welches das Applet enthält (in der Regel eine HTML-Seite) und den anderen Applets in dem gleichen Dokument. Die Methoden in dieser Schnittstelle können für die Informationsbeschaffung über die Umgebung eines Applets verwendet werden. Die Schnittstelle `public abstract interface AppletStub` hilft bei der Erstellung eines AppletStubs. Der AppletStub dient dazu, als Interface zwischen dem Applet und der Browser-/Appletviewer-Umgebung zu fungieren.

Eine seit dem JDK 1.2 neu eingeführte Subklasse der Klasse `Applet` ist `JApplet` (`public class JApplet extends Applet implements Accessible, RootPaneContainer`), die mittlerweile im Paket `javax.swing` zu finden ist. Sie erweitert die Applet-Fähigkeiten von Java in Bezug auf die Swing-Funktionalität. Diese Subklasse übernimmt zwar die meisten Elemente und Methoden aus der Superklasse, ist jedoch in einigen Bereichen leicht inkompatibel mit `java.applet.Applet`. `JApplet` beinhaltet ein `JRootPane` als einziges Kind. Die Methode `contentPane()` sollte das Elternelement von jedem Kind (darunter ist beispielsweise eine Komponente wie eine Schaltfläche zu verstehen) von `JApplet` sein. Dies ist verschieden zu `java.applet.Applet`. Dort ist es beispielsweise möglich, ein Kind zu einem Applet wie folgt hinzuzufügen:

```
applet.add(child);
```

Mit dem `JApplet` muss statt dessen der `contentPane()` verwendet werden:

```
applet.getContentPane().add(child);
```

Das Gleiche gilt für die Arbeit mit LayoutManager, das Entfernen von Komponenten und ähnliche Aktionen. Alle diese Methoden sollten innerhalb des Swingkonzeptes normalerweise auf `contentPane()` statt `JApplet` selbst angewandt werden. `contentPane()` wird und darf nie `null` sein (Auswurf einer Exception). Default-Einstellung bei der Verwendung von `contentPane()` ist der BorderLayoutmanager.

JDK-Version

Seit 1.0

Warnungen

Falls am Ende eines Applets Aufräumarbeiten notwendig sind, kann die allgemeinere `finalize()`-Methode anstelle von `destroy()` nicht (!) verwendet werden.

Enthaltene Klassen

```
public class Applet extends Panel
```

Enthaltene Schnittstellen

```
public abstract interface AudioClip
public abstract interface AppletContext
public abstract interface AppletStub
```

Enthaltene Ausnahmen

(keine)

java.awt

Die wesentlichen Elemente zur Gestaltung einer grafischen Benutzeroberfläche im Rahmen des Abstract Windowing Toolkit (AWT).

Beschreibung

Das Paket beinhaltet zusammen mit seinen Unterpaketen alle wesentlichen Klassen und Schnittstellen zur Erstellung einer Benutzerschnittstelle sowie Zeichnen- und Bildausgabeoperationen.

Anwendung

Das Paket ist eines der umfangreichsten Pakete der Java-Plattform und Grundlage zur Gestaltung einer grafischen Schnittstelle für die Anwenderkommunikation. Dies beginnt bei der Bereitstellung von Komponenten wie Schaltflächen oder Menüs, geht über Manager für das Layout der grafischen Oberfläche bis hin zur Ereignisbehandlung. Daneben sind zahlreiche Elemente enthalten, welche die Ausgabe von Multimedia unterstützen. Unterstützt wird das Paket von zahlreichen Unterpaketen.

Die Wurzel von fast allen AWT-Komponenten ist die in dem Paket enthaltene Klasse `Component`. Das AWT beinhaltet zur Kommunikation mit dem Anwender inzwischen alle wesentlichen Elemente zur Erstellung einer plattformunabhängigen Benutzerschnittstelle. Die Komponenten kann man dabei nach ihrer Funktion in drei logische Bereiche unterteilen:

- Aktive Komponenten einer Benutzeroberfläche: Dazu zählen alle typischen Elemente einer Benutzeroberfläche (etwa Schaltflächen, Kontrollfelder,...). Diese werden durch Klassen wie `Button`, `Checkbox`, `Choise` usw. realisiert.
- Zeichenbereiche: Dabei handelt es sich um eine Fläche auf dem Bildschirm, auf der in der Regel Zeichnungen ausgegeben werden.
- Fensterkomponenten: Dies sind alle Komponenten, welche sich um den Aufbau und die Struktur der Fenster kümmern. Also Bildlaufleisten, Rahmen, Menüleisten und Dialogfelder. Klassen wie `Dialog`, `Frame` oder `Window` bieten dafür die Umsetzung. Diese Fensterkomponenten trennt man logisch von den aktiven Komponenten der Benutzeroberfläche, da sie logisch unabhängig von der integrierten Anwendung betätigt werden (das Verschieben von Fensterinhalt über Bildlaufleisten beispielsweise löst noch keine Aktion in der Applikation aus).

Ein wesentliches Konzept der Java-Oberflächengestaltung sind Layoutmanager, welche sich relativ selbständig um das konkrete Aussehen der Benutzeroberfläche kümmern. Layoutmanager ordnen Komponenten entsprechend ihren Vorgaben in dem Container an und legen ihre Größe fest. Dabei nehmen sie Rücksicht auf die Größe des Container-Fensters und auf Besonderheiten der Plattform. Das Interface

`public interface LayoutManager extends Object` definiert dazu die zentralen Klassen für das Layout von Containern und ist Grundlage für die über die Klassen `GridLayout`, `FlowLayout`, `ViewportLayout` und `ScrollPaneLayout` realisierten Layoutmanager. Schnittstellen wie `public interface LayoutManager2 extends LayoutManager` erweitern das Interface für zahlreiche andere Layoutklassen (insbesondere in Bezug auf das Swingkonzept) wie `CardLayout`, `GridBagLayout`, `BorderLayout`, `OverlayLayout`, `BoxLayout` oder `JRootPane.RootLayout`.

Seit der Version 1.0 haben sich gerade im AWT massive Erweiterungen und Veränderungen ergeben. Insbesondere hat das Abstract Windowing Toolkit bereits in der Version 1.1 sehr weitreichende Erweiterungen erfahren. Darunter fallen im Wesentlichen eine einheitliche Druckerschnittstelle zum plattformunabhängigen Drucken, schnelleres und einfacheres Scrolling (beziehungsweise die prinzipiell schnellere Reaktion auf verschiedene Events), bessere Grafikmöglichkeiten sowie flexiblere Font-Unterstützung. Eine Unterstützung von Pop-up-Menüs und der Zwischenablage waren weitere wichtige Erweiterungen der ersten AWT-Überarbeitung. Das neueste API hat noch einmal zugelegt und viele weitere neue Funktionalitäten und Erweiterungen folgen lassen. Die Java Foundation Classes (JFC) beinhalten nun Java 2D, UI (UI steht für User Interface – Benutzerschnittstelle), Components (Swing Package), Zugriffsmöglichkeiten auf Fremdtechnologien (Accessibility), Drag&Drop und Application Services. Ein ganz wesentlicher Aspekt ist das Swing-Konzept, mit dem eine Erweiterung des bisherigen Aussehens der Oberfläche geschaffen wurde. Dieses Konzept implementiert einen neuen Satz von GUI-Komponenten mit anpassungsfähigem Look and Feel. Swing ist vollständig in 100% purem Java implementiert und basiert auf dem JDK 1.1 Lightweight UI Framework. Das Aussehen und die Reaktionen von GUI-Komponenten passen sich auf Wunsch des Entwicklers automatisch an jede unterstützte Betriebssystemplattform (Windows, Solaris, Macintosh) an und erweitern die bisherigen AWT-Möglichkeiten um einen Satz von Oberflächen-Interaktionselementen (Baumansichten, Listboxen, usw.).

Zusammen mit der optischen Gestaltung der Oberfläche über die AWT-Komponenten ist auch die Reaktion auf Ereignisse auf der Oberfläche (etwa ein Klick auf eine Schaltfläche oder die Betätigung einer Taste) zu sehen. Dieses sogenannte Event-handling war unter dem AWT 1.0 sehr starr ausgelegt und wurde oft kritisiert. Um abwärtskompatibel zu bleiben, enthält das AWT die Klasse `java.awt.Event` zur Unterstützung des Eventmodells 1.0. In der Version 1.1 wurde es jedoch völlig überarbeitet und damit wesentlich flexibler. Das 1.1-Eventmodell arbeitet mit sogenannten Delegates. Dies sind spezielle Objekte, welche die Delegierten von anderen Objekten sind und an bestimmte Methoden geschickt werden, die das Delegate dann abarbeiten. Zentraler Aspekt in dem aktuellen Eventhandling ist die Klasse `public class AWTEvent extends EventObject`, welche die Rootklasse von allen Events ist, die unter dem aktuellen Eventmodell durch AWT-Klassen erzeugt werden. Direkte Subklassen davon sind `ActionEvent`, `AdjustmentEvent`,

AncestorEvent, ComponentEvent, InputMethodEvent, InternalFrameEvent, ItemEvent und TextEvent. Die Klasse java.awt.AWTEventMulticaster sorgt dafür, dass Events aus dem Package java.awt.event an die jeweils entsprechenden Listener weitergereicht werden. Dort befinden sich für das Eventhandling primäre Methoden zum Registrieren, Hinzufügen oder Löschen von Listenern beziehungsweise der Reaktion darauf (wie

```
public static ComponentListener add(ComponentListener a, Component-Listener b),
public static ContainerListener add(ContainerListener a, ContainerListener b),
public static FocusListener add(FocusListener a, FocusListener b), public
static KeyListener add(KeyListener a, KeyListener b), public static
MouseListener add(MouseListener a, MouseListener b), public static
MouseMotionListener add(MouseMotionListener a, Mouse-MotionListener b), public
static WindowListener add(WindowListener a, WindowListener b), public static
ActionListener add(ActionListener a, ActionListener b), public static
ItemListener add(ItemListener a, ItemListener b), public static
AdjustmentListener add(AdjustmentListener a, Adjustment-Listener b), public
static TextListener add(TextListener a, TextListener b), public void
componentAdded(ContainerEvent e), public void componentHidden(ComponentEvent
e), public void componentMoved(ComponentEvent e), public void
componentRemoved(ComponentEvent e),public static ComponentListener
remove(ComponentListener l, ComponentListener old), public static
containerListener remove(ContainerListener l, ContainerListener old), public
static FocusListener remove(FocusListener l, FocusListener old), public static
KeyListener remove(KeyListener l, KeyListener old), public static MouseListener
remove(MouseListener l, MouseListener old)
```

), sowie zahlreiche Methoden zur Reaktion auf Ereignisse (etwa

```
public void focusGained(FocusEvent e), public void
focusLost(FocusEvent e), public void keyPressed(KeyEvent e), public void
keyTyped(KeyEvent e), public void mouseClicked(MouseEvent e), public void
mouseDragged(MouseEvent e), public void mouseEntered(MouseEvent e), public void
mouseExited(MouseEvent e), public void mouseMoved(MouseEvent e), public void
mousePressed(MouseEvent e), public void mouseReleased(MouseEvent e), public
void textValueChanged(TextEvent e), public void windowActivated(WindowEvent e),
public void windowClosed(WindowEvent e), public void windowClosing(WindowEvent
e), public void windowDeactivated(WindowEvent e), public void
windowDeiconified(WindowEvent e), public void windowIconified(WindowEvent e),
public void windowOpened(WindowEvent e)
```

).

Auch für die konkrete optische Gestaltung der Oberfläche bietet das AWT zahlreiche Klassen und Schnittstellen. Die Klasse `public class Color extends Object implements Paint, Serializable` beispielsweise stellt RGB-Farben dar und beinhaltet dazu unter anderem zahlreiche Konstanten für die gängigen Farben (etwa `public final static Color black` für die Farbe Schwarz, `public final static Color blue` für die Farbe Blau oder `public final static Color gray` für Grau). Die allgemeine grafische Gestaltung des AWT basiert auf der abstrakten Klasse `public abstract class Graphics extends Object`, die als systemunabhängige Basis für alle grafischen Kontexte fungiert. Sie beinhaltet unter anderen Methoden zum Zeichnen von einfachen geometrischen Figuren. Noch leistungsfähiger, aber an neuere Java-Plattformen gebunden, ist die Klasse `public abstract class Graphics2D extends Graphics`, welche

von fundamentaler Bedeutung für die 2D-Grafik unter Java ist. Sie erweitert die originale `java.awt.Graphics`-Klasse und bietet erhebliche grafische Verbesserungen. Die abstrakte Klasse `java.awt.Image` hingegen ist die Grundlage der Bildverarbeitung im AWT.

Auch die individuelle Gestaltung des Cursors je nach Kontext ist unter dem AWT kein kompliziertes Thema. Die Klasse `public class Cursor extends Object implements Serializable` unterstützt die Behandlung der Bitmap-Repräsentation des Cursors. Auch sie enthält neben diversen Methoden zahlreiche Konstanten für einen komfortablen Umgang damit (etwa `public static final int DEFAULT_CURSOR`, `public static final int CROSSHAIR_CURSOR`, `public static final int HAND_CURSOR`, `public static final int MOVE_CURSOR`).

JDK-Version

Seit 1.0

Warnungen

Das AWT-Modell wurde zwar bereits in der Java-Version 1.1 gegenüber seinen Vorgängern erheblich verändert. Das davor gültige AWT-Modell (inklusive dem alten Eventhandling) hat jedoch immer noch erhebliche Bedeutung. Ein nicht zu unterschätzendes Problem des neueren AWT und des aktuellen Eventhandlings ist der gänzlich unterschiedliche Aufbau zu dem Modell 1.0. Komplexe Entwicklungen, die bereits nach dem alten Verfahren erstellt wurden, müssten mit viel Aufwand auf das neue Konzept umgestellt werden, ohne dass für einen eventuellen Kunden der Aufwand offensichtlich wird[17]. Des Weiteren unterstützen noch nicht alle Browser und Interpreterplattformen das neue Eventmodell und die damit assoziierten neuen Methoden beziehungsweise neuen Varianten von bereits vorhandenen Methoden.

Enthaltene Klassen

```
public final class AlphaComposite extends Object implements Composite
public abstract class AWTEvent extends EventObject
public class AWTEventMulticaster extends Object implements ComponentListener,
ContainerListener, FocusListener, KeyListener, MouseListener, MouseMotionListener,
WindowListener, ActionListener, ItemListener, AdjustmentListener, TextListener,
InputMethodListener
public final class AWTPermission extends BasicPermission
public class BasicStroke extends Object implements Stroke
public class BorderLayout extends Object implements LayoutManager2, Serializable
public class Button extends Component
public class Canvas extends Component
public class CardLayout extends Object implements LayoutManager2, Serializable
```

17. Zudem - never touch a runing system.

```
public class Checkbox extends Component implements ItemSelectable
public class CheckboxGroup extends Object implements Serializable
public class CheckboxMenuItem extends MenuItem implements ItemSelectable
public class Choice extends Component implements ItemSelectable
public class Color extends Object implements Paint, Serializable
public abstract class Component extends Object implements ImageObserver,
MenuContainer, Serializable
public final class ComponentOrientation extends Object implements Serializable
public class Container extends Component
public class Cursor extends Object implements Serializable
public class Dialog extends Window
public class Dimension extends Dimension2D implements Serializable
public class Event extends Object implements Serializable
public class EventQueue extends Object
public class FileDialog extends Dialog
public class FlowLayout extends Object implements LayoutManager, Serializable
public class Font extends Object implements Serializable
public abstract class FontMetrics extends Object implements Serializable
public class Frame extends Window implements MenuContainer
public class GradientPaint extends Object implements Paint
public abstract class Graphics extends Object
public abstract class Graphics2D extends Graphics
public abstract class GraphicsConfigTemplate extends Object implements
Serializable
public abstract class GraphicsConfiguration extends Object
public abstract class GraphicsDevice extends Object
public abstract class GraphicsEnvironment extends Object
public class GridBagConstraints extends Object implements Cloneable, Serializable
public class GridBagLayout extends Object implements LayoutManager2, Serializable
public class GridLayout extends Object implements LayoutManager, Serializable
public abstract class Image extends Object
public class Insets extends Object implements Cloneable, Serializable
public class Label extends Component
public class List extends Component implements ItemSelectable
public class MediaTracker extends Object implements Serializable
public class Menu extends MenuItem implements MenuContainer
public class MenuBar extends MenuComponent implements MenuContainer
public abstract class MenuComponent extends Object implements Serializable
public class MenuItem extends MenuComponent
public class MenuShortcut extends Object implements Serializable
public class Panel extends Container
public class Point extends Point2D implements Serializable
public class Polygon extends Object implements Shape, Serializable
public class PopupMenu extends Menu
public abstract class PrintJob extends Object
public class Rectangle extends Rectangle2D implements Shape, Serializable
public class RenderingHints extends Object implements Map, Cloneable
public abstract static class RenderingHints.Key extends Object
public class Scrollbar extends Component implements Adjustable
public class ScrollPane extends Container
public final class SystemColor extends Color implements Serializable
```

```
public class TextArea extends TextComponent
public class TextComponent extends Component
public class TextField extends TextComponent
public class TexturePaint extends Object implements Paint
public abstract class Toolkit extends Object
public class Window extends Container
```

Enthaltene Schnittstellen

```
public abstract interface ActiveEvent
public abstract interface Adjustable
public abstract interface Composite
public abstract interface CompositeContext
public abstract interface ItemSelectable
public abstract interface LayoutManager
public abstract interface LayoutManager2 extends LayoutManager
public abstract interface MenuContainer
public abstract interface Paint extends Transparency
public abstract interface PaintContext
public abstract interface PrintGraphics
public abstract interface Shape
public abstract interface Stroke
public abstract interface Transparency
```

Enthaltene Ausnahmen

```
public class AWTException extends Exception
public class IllegalComponentStateException extends IllegalStateException
```

Enthaltene Errors

```
public class AWTError extends Error
```

java.awt.color

Erweiterte Unterstützung für Farbräume.

Beschreibung

Das Paket beinhaltet Klassen zur erweiterten Unterstützung von Farben unter grafischen Oberflächen im Rahmen des AWT.

Anwendung

Dieses Paket beschreibt verschiedene Systeme, um Farben zu identifizieren und die Konvertierung zwischen den Systemen zu ermöglichen. Es beinhaltet eine Implementation von einem Farbraum, der auf der International Color Consortium (ICC) Profile Format Specification, Version 3.4, August 15, 1997 basiert. Daneben beinhaltet das Paket Farbprofile, die auf der besagten ICC Profile Format Specification basieren.

JDK-Version

Seit 1.2

Enthaltene Klassen

```
public abstract class ColorSpace extends Object
public class ICC_ColorSpace extends ColorSpace
public class ICC_Profile extends Object
public class ICC_ProfileGray extends ICC_Profile
public class ICC_ProfileRGB extends ICC_Profile
```

Enthaltene Schnittstellen

(keine)

Enthaltene Ausnahmen

```
public class CMMException extends RuntimeException
public class ProfileDataException extends RuntimeException
```

java.awt.datatransfer

Unterstützung von Datentransfer.

Beschreibung

Das Paket beinhaltet Schnittstellen und Klassen zur Unterstützung von Datentransferaktionen innerhalb und zwischen Applikationen.

Anwendung

Dieses Paket erlaubt den Datenaustausch von Informationen innerhalb eines oder verschiedener Programme mittels `cut`, `copy` und `paste`. Dazu wird im Wesentlichen das bekannte Clipboard beziehungsweise Zwischenablage (ein Speicherbereich) genutzt (in Form einer eigenen Klasse), das nahezu alle datenbasierenden Programme unterstützen.

Daneben befindet sich in dem Paket die Schnittstelle `Transferable`, durch dessen Implementation sich ein Objekt als transferierbar definieren kann.

JDK-Version

Seit 1.1

Enthaltene Klassen

```
public class Clipboard extends Object
public class DataFlavor extends Object implements Externalizable, Cloneable
public class StringSelection extends Object implements Transferable,
ClipboardOwner
public final class SystemFlavorMap extends Object implements FlavorMap
```

Enthaltene Schnittstellen

```
public abstract interface ClipboardOwner
public abstract interface FlavorMap
public abstract interface Transferable
```

Enthaltene Ausnahmen

```
public class UnsupportedFlavorException extends Exception
```

java.awt.dnd

Unterstützung von Datentransfer per Drag&Drop.

Beschreibung

Das Paket beinhaltet Schnittstellen und Klassen zur Behandlung beider Seiten von einer Drag&Drop-Operation.

Anwendung

Dieses Paket erlaubt den Datenaustausch von Informationen innerhalb eines oder verschiedener Programme mittles Drag&Drop. Es definiert Klassen für den Drag-Source (den Quellbereich der Aktion) und das Drop-Target (den Zielbereich der Aktion) sowie Events für die Transferaktion selbst (inklusive einem visuellen Feedback für den Anwender der Operation – beispielsweise die Veränderung der Mauszeigerform).

JDK-Version

Seit 1.2

Enthaltene Klassen

```
public final class DnDConstants extends Object
public class DragGestureEvent extends EventObject
public abstract class DragGestureRecognizer extends Object
public class DragSource extends Object
public class DragSourceContext extends Object implements DragSourceListener
public class DragSourceDragEvent extends DragSourceEvent
public class DragSourceDropEvent extends DragSourceEvent
public class DragSourceEvent extends EventObject
public class DropTarget extends Object implements DropTargetListener, Serializable
protected static class DropTarget.DropTargetAutoScroller extends Object implements
ActionListener
public class DropTargetContext extends Object
public class DropTargetDragEvent extends DropTargetEvent
public class DropTargetDropEvent extends DropTargetEvent
public class DropTargetEvent extends EventObject
public abstract class MouseDragGestureRecognizer extends DragGestureRecognizer
implements MouseListener, MouseMotionListener
```

Enthaltene Schnittstellen

```
public abstract interface Autoscroll
public abstract interface DragGestureListener extends EventListener
public abstract interface DragSourceListener extends EventListener
public abstract interface DropTargetListener extends EventListener
```

Enthaltene Ausnahmen

```
public class InvalidDnDOperationException extends IllegalStateException
```

java.awt.event

Das Basispaket für das 1.1-Eventmodell.

Beschreibung

Das Paket beinhaltet Schnittstellen und Klassen zur Behandlung von verschiedenen Typen von Ereignissen, welche von AWT-Komponenten im Rahmen des 1.1-Eventmodells ausgelöst werden.

Anwendung

Seit dem JDK 1.1 werden von AWT-Komponenten Events ausgelöst, welche von sogenannten Event-Listenern (spezielle Objekte, die Ereignisse warten und darauf reagieren) registiert werden. Dabei werden zahlreiche Informationen über das auslösende Event an den Listener übergeben. Das Paket definiert sowohl die Events und die Event-Listener (für alle notwendigen Maus- und Tastaturereignisse mit eigenen Klassen), aber auch Event-Listeneradapter, welche als ergänzende Klassen zu verstehen sind, die eine Erstellung von neuen Event-Listenern erlauben.

JDK-Version

Seit 1.1

Warnungen

Das Ereignismodell des AWT wurde zwar bereits in der Java-Version 1.1 gegenüber seinen Vorgängern erheblich verändert (zusammen mit dem vollständigen AWT-Umfeld). Das davor gültige Eventhandling beziehungsweise das vollständige AWT-Umfeld hat jedoch immer noch erhebliche Bedeutung. Komplexe Entwicklungen, die nach dem alten Verfahren erstellt wurden, müssten mit viel Aufwand auf das neue Konzept umgestellt werden, ohne dass der Aufwand für einen Betrachter offensichtlich wird. Des Weiteren unterstützen noch nicht alle Browser und Interpreterplattformen das neue Eventmodell und die damit assoziierten neuen Methoden beziehungsweise neuen Varianten von bereits vorhandenen Methoden.

Enthaltene Klassen

```
public class ActionEvent extends AWTEvent
public class AdjustmentEvent extends AWTEvent
public abstract class ComponentAdapter extends Object implements ComponentListener
public class ComponentEvent extends AWTEvent
public abstract class ContainerAdapter extends Object implements ContainerListener
public class ContainerEvent extends ComponentEvent
public abstract class FocusAdapter extends Object implements FocusListener
public class FocusEvent extends ComponentEvent
public abstract class InputEvent extends ComponentEvent
public class InputMethodEvent extends AWTEvent
```

```
public class InvocationEvent extends AWTEvent implements ActiveEvent
public class ItemEvent extends AWTEvent
public abstract class KeyAdapter extends Object implements KeyListener
public class KeyEvent extends InputEvent
public abstract class MouseAdapter extends Object implements MouseListener
public class MouseEvent extends InputEvent
public abstract class MouseMotionAdapter extends Object implements
MouseMotionListener
public class PaintEvent extends ComponentEvent
public class TextEvent extends AWTEvent
public abstract class WindowAdapter extends Object implements WindowListener
public class WindowEvent extends ComponentEvent
```

Enthaltene Schnittstellen

```
public abstract interface ActionListener extends EventListener
public abstract interface AdjustmentListener extends EventListener
public abstract interface AWTEventListener extends EventListener
public abstract interface ComponentListener extends EventListener
public abstract interface ContainerListener extends EventListener
public abstract interface FocusListener extends EventListener
public abstract interface InputMethodListener extends EventListener
public abstract interface ItemListener extends EventListener
public abstract interface KeyListener extends EventListener
public abstract interface MouseListener extends EventListener
public abstract interface MouseMotionListener extends EventListener
public abstract interface TextListener extends EventListener
public abstract interface WindowListener extends EventListener
```

Enthaltene Ausnahmen

```
public class InvalidDnDOperationException extends IllegalStateException
```

java.awt.font

Erweiterte Unterstützung von Schriften.

Beschreibung

Das Paket beinhaltet Schnittstellen und Klassen zur Unterstützung von verschiedenen Typen von Schriften und deren Behandlung.

Anwendung

Dieses Paket unterstützt die Zusammenstellung, Anzeige und individuelle Gestaltung von Schriftarten. Dabei stellt das Paket so ziemlich alles zur Verfügung, was man von einer solchen Zusammenstellung, Anzeige und individuellen Gestaltung für Schriftarten erwarten kann:

- Stilbeeinflussungen
- Metriken
- Grafische Ausprägungen
- Liniengestaltung
- Transformationen

und vieles mehr.

JDK-Version

Seit 1.2

Warnungen

Das Paket wurde gegenüber seinen Vorgängern in den Betaversionen erheblich verändert (was leider von Sun unzureichend dokumentiert wurde). So finden Sie im Final beispielsweise die folgenden Klassen nicht mehr, welche in den Betaversionen vorhanden waren:

```
GlyphSet
StyledString
StyledStringIterator
```

Enthaltene Klassen

```
public class FontRenderContext extends Object
public final class GlyphJustificationInfo extends Object
public final class GlyphMetrics extends Object
public abstract class GlyphVector extends Object implements Cloneable
public abstract class GraphicAttribute extends Object
public final class ImageGraphicAttribute extends GraphicAttribute
public final class LineBreakMeasurer extends Object
```

```
public abstract class LineMetrics extends Object
public final class ShapeGraphicAttribute extends GraphicAttribute
public final class TextAttribute extends AttributedCharacterIterator.Attribute
public final class TextHitInfo extends Object
public final class TextLayout extends Object implements Cloneable
public static class TextLayout.CaretPolicy extends Object
public static final class TextLine.TextLineMetrics extends Object
public final class TransformAttribute extends Object implements Serializable
```

Enthaltene Schnittstellen

```
public abstract interface MultipleMaster
public abstract interface OpenType
```

Enthaltene Ausnahmen

(keine)

java.awt.geom

Erweiternde Unterstützung von 2D-Geometrie unter Java.

Beschreibung

Das Paket beinhaltet eine Schnittstelle und zahlreiche Klassen zur Unterstützung von Java-2D-Klassen zur Definition und Gewährleistung von Operationen mit Objekten unter einer zweidimensionalen Geometrie.

Anwendung

Java unterstützt bereits mit dem Paket `java.awt` zweidimensionale Grafik. In diesem Paket werden diese Grundlagen noch erweitert. Die Unterstützung der Definition und Gewährleistung von Operationen mit Objekten unter einer zweidimensionalen Geometrie folgt den üblichen mathematischen 2D-Geometrieregeln. Einige wichtige Features von dem Paket sind folgende:

- Zahlreiche Klassen für die Manipulation von geometrischen Vorgängen. Etwa affine Transformationen und das `PathIterator`-Interface, welches in allen Objekten implementiert ist, die eine Form beschreiben.
- Einige Standardklassen für Formen wie `CubicCurve2D`, `Ellipse2D`, `Line2D`, `Rectangle2D`, und `GeneralShape`.
- Die `Area`-Klasse, welche Mechanismen bereitstellt für Operationen zwischen geometrischen Objekten (etwa das Überlagern oder das Schneiden).

JDK-Version

Seit 1.2

Enthaltene Klassen

```
public class AffineTransform extends Object implements Cloneable, Serializable
public abstract class Arc2D extends RectangularShape
public static class Arc2D.Double extends Arc2D
public static class Arc2D.Float extends Arc2D
public class Area extends Object implements Shape, Cloneable
public abstract class CubicCurve2D extends Object implements Shape, Cloneable
public static class CubicCurve2D.Double extends CubicCurve2D
public static class CubicCurve2D.Float extends CubicCurve2D
public abstract class Dimension2D extends Object implements Cloneable
public abstract class Ellipse2D extends RectangularShape
public static class Ellipse2D.Double extends Ellipse2D
public static class Ellipse2D.Float extends Ellipse2D
public class FlatteningPathIterator extends Object implements PathIterator
public final class GeneralPath extends Object implements Shape, Cloneable
public abstract class Line2D extends Object implements Shape, Cloneable
public static class Line2D.Double extends Line2D
```

```
public static class Line2D.Float extends Line2D
public abstract class Point2D extends Object implements Cloneable
public static class Point2D.Double extends Point2D
public static class Point2D.Float extends Point2D
public abstract class QuadCurve2D extends Object implements Shape, Cloneable
public static class QuadCurve2D.Double extends QuadCurve2D
public static class QuadCurve2D.Float extends QuadCurve2D
public abstract class Rectangle2D extends RectangularShape
public static class Rectangle2D.Double extends Rectangle2D
public static class Rectangle2D.Float extends Rectangle2D
public abstract class RectangularShape extends Object implements Shape, Cloneable
public abstract class RoundRectangle2D extends RectangularShape
public static class RoundRectangle2D.Double extends RoundRectangle2D
public static class RoundRectangle2D.Float extends RoundRectangle2D
```

Enthaltene Schnittstellen

```
public abstract interface PathIterator
```

Enthaltene Ausnahmen

```
public class IllegalPathStateException extends RuntimeException
public class NoninvertibleTransformException extends Exception
```

java.awt.im

Unterstützung von Zeichen aus dem asiatischen Raum.

Beschreibung

Eine Schnittstelle und zahlreiche Klassen für das Eingabeframework von Java.

Anwendung

Das Paket bietet Unterstützung von dem Eingabeframework von Java. Dieses Framework erlaubt es allen textverarbeitenden Komponenenten, japanische, chinesische oder koreanische Texteingaben über normale Eingabemethoden zu empfangen. Dies bedeutet, dass eine Eingabemethode Tausende von verschiedenen Zeichen über Tastenkombinationen empfangen kann. Verwandte Klassen im Paket `java.awt.event` können dies nutzen.

JDK-Version

Seit 1.2

Enthaltene Klassen

```
public class InputContext extends Object
public class InputMethodHighlight extends Object
public final class InputSubset extends Character.Subset
```

Enthaltene Schnittstellen

```
public abstract interface InputMethodRequests
```

Enthaltene Ausnahmen

(keine)

java.awt.image

Erweiternde Unterstützung für Bilder im Rahmen des AWT.

Beschreibung

Hierbei handelt es sich um ein Paket, um unter dem AWT Bitmap-Bilder zu handhaben. Dazu werden Klassen und Schnittstellen bereitgestellt, welche das Erstellen und Modifizieren von Bildern erlauben.

Anwendung

Bereits im Grundpaket `java.awt` lassen sich Bilder handhaben. Diese grundlegenden Techniken werden hier weiter ausgebaut. Beim Erstellen und Modifizieren von Bildern werden diese über ein Streamingframework behandelt, welches einen Imageproducer (ein Objekt, das Bilder produziert), optionale Bildfilter und einen Imageconsumer beinhaltet. Dieses Framework bietet zahlreiche Beeinflussungsmöglichkeiten für Bilder. Sowohl wärend der Erstellung selbst, als auch im Nachhinein.

JDK-Version

Seit 1.0

Warnungen

Diese Paket wurde gegenüber den Vorgängern (auch noch den Betaversionen 1.2) erheblich umstrukturiert. So fehlen zahlreiche Schnittstellen und Klassen, welche dort noch vorhanden waren. Dies wird teilweise nur unvollständig dokumentiert.

Enthaltene Klassen

```
public class AffineTransformOp extends Object implements BufferedImageOp, RasterOp
public class AreaAveragingScaleFilter extends ReplicateScaleFilter
public class BandCombineOp extends Object implements RasterOp
public final class BandedSampleModel extends ComponentSampleModel
public class BufferedImage extends Image implements WritableRenderedImage
public class BufferedImageFilter extends ImageFilter implements Cloneable
public class ByteLookupTable extends LookupTable
public class ColorConvertOp extends Object implements BufferedImageOp, RasterOp
public abstract class ColorModel extends Object implements Transparency
public class ComponentColorModel extends ColorModel
public class ComponentSampleModel extends SampleModel
public class ConvolveOp extends Object implements BufferedImageOp, RasterOp
public class CropImageFilter extends ImageFilter
public abstract class DataBuffer extends Object
public final class DataBufferByte extends DataBuffer
public final class DataBufferInt extends DataBuffer
public final class DataBufferShort extends DataBuffer
```

```
public final class DataBufferUShort extends DataBuffer
public class DirectColorModel extends PackedColorModel
public class FilteredImageSource extends Object implements ImageProducer
public class ImageFilter extends Object implements ImageConsumer, Cloneable
public class IndexColorModel extends ColorModel
public class Kernel extends Object implements Cloneable
public class LookupOp extends Object implements BufferedImageOp, RasterOp
public abstract class LookupTable extends Object
public class MemoryImageSource extends Object implements ImageProducer
public class MultiPixelPackedSampleModel extends SampleModel
public abstract class PackedColorModel extends ColorModel
public class PixelGrabber extends Object implements ImageConsumer
public class PixelInterleavedSampleModel extends ComponentSampleModel
public class Raster extends Object
public class ReplicateScaleFilter extends ImageFilter
public class RescaleOp extends Object implements BufferedImageOp, RasterOp
public abstract class RGBImageFilter extends ImageFilter
public abstract class SampleModel extends Object
public class ShortLookupTable extends LookupTable
public class SinglePixelPackedSampleModel extends SampleModel
public class WritableRaster extends Raster
```

Enthaltene Schnittstellen

```
public abstract interface BufferedImageOp
public abstract interface ImageConsumer
public abstract interface ImageObserver
public abstract interface ImageProducer
public abstract interface RasterOp
public abstract interface RenderedImage
public abstract interface TileObserver
public abstract interface WritableRenderedImage extends RenderedImage
```

Enthaltene Ausnahmen

```
public class ImagingOpException extends RuntimeException
public class RasterFormatException extends RuntimeException
```

java.awt.image.renderable

Weitergehende Bildunterstützung im Rahmen des AWT.

Beschreibung

Ein Paket mit unterstützenden Klassen und Schnittstellen für die Produktion von Rendering(Wiedergabe)-unabhängigen Bildern unter dem AWT.

Anwendung

Um vom Wiedergabemedium und den konkreten Bedingungen der Darstellungsumgebung unabhängige Bilder (samt den Operationen damit) zu generieren, werden in den Klassen und Schnittstellen Funktionalitäten bereitgestellt, welche zahlreiche Hintergrundinformationen über ein Bild speichern und nutzbar machen. Damit lässt sich sicherstellen, dass Bilder unter jedweder Umgebung (soweit technisch möglich) identisch aussehen oder je nach Umgebung ein spezifisches, im Bild fest vorgegebenes Aussehen bekommen.

JDK-Version

Seit 1.2

Enthaltene Klassen

```
public class ParameterBlock extends Object implements Cloneable, Serializable
public class RenderableImageOp extends Object implements RenderableImage
public class RenderableImageProducer extends Object implements ImageProducer,
Runnable
public class RenderContext extends Object implements Cloneable
```

Enthaltene Schnittstellen

```
public abstract interface ContextualRenderedImageFactory extends
RenderedImageFactory
public abstract interface RenderableImage
public abstract interface RenderedImageFactory
```

Enthaltene Ausnahmen

(keine)

java.awt.print

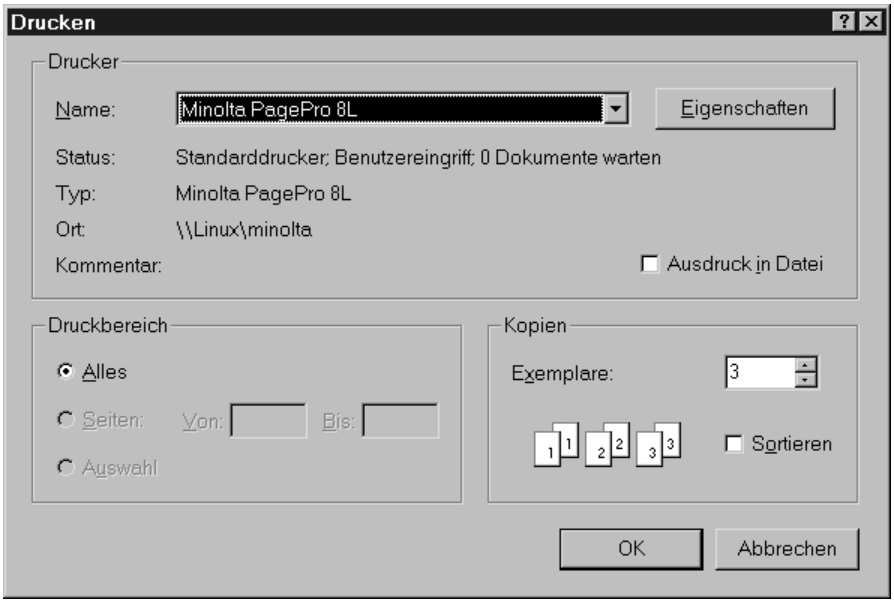
Allgemeine Druckunterstützung im AWT und Java im Allgemeinen.

Beschreibung

Das Paket stellt Klassen und Schnittstellen zur Verfügung, um unter Java auf die jeweiligen Druckmöglichkeiten einer Plattform zugreifen zu können.

Anwendung

Das Druck-Package stellt Schnittstellen und Klassen zur Verfügung, welche eine vollständige Kontrolle eines Druckauftrags auf jeder Java unterstützenden Plattform gewährleisten (insbesondere mit dialogbasierender Auswahlmöglichkeit durch den Anwender). Java-typisch greift man damit auf einer hohen Ebene auf die Druckroutinen des jeweiligen Systems zu und kann die dort bereitgestellten Standards nutzen. Das Paket stellt sowohl Listen der zu druckenden Seiten (mit Informationen über eine Seite – etwa den druckbaren Bereich) zur Verfügung (die Klasse `Book`) als auch Informationen über das Seitenformat – im Wesentlichen die Größe und Ausrichtung von Seiten (die Klasse `PageFormat`), die physikalische Charakterisierung des Druckpapiers (die Klasse `Paper`) bis hin zur Kontrolle des konkreten Druckauftrags (die Klasse `PrinterJob`). Die verschiedenen Ausnahmen erlauben es, qualifiziert auf Rückmeldungen durch den Drucker zu reagieren.



Im JDK 1.2 lässt sich einfach und komfortabel der Standarddruckvorgang des Betriebssystem aufrufen.

JDK-Version

Seit 1.2

Warnungen

Das Paket ist besonders deshalb bemerkenswert, weil in den bisherigen Versionen von Java die Druckunterstützung nur sehr mangelhaft gelöst war. Druckunterstützung ist erst seit dem JDK 1.2 (im Wesentlichen über dieses Paket) unter Java befriedigend gelöst. Wenn mittels einer älteren Version des JDK gedruckt werden soll, müssen ziemlich umständliche Wege beschritten werden.

Enthaltene Klassen

```
public class Book extends Object implements Pageable
public class PageFormat extends Object implements Cloneable
public class Paper extends Object implements Cloneable
public abstract class PrinterJob extends Object
```

Enthaltene Schnittstellen

```
public abstract interface Pageable
public abstract interface Printable
public abstract interface PrinterGraphics
```

Enthaltene Ausnahmen

```
public class PrinterAbortException extends PrinterException
public class PrinterException extends Exception
public class PrinterIOException extends PrinterException
```

java.beans

Die Unterstützung des Beans-Konzeptes.

Beschreibung

Hierbei handelt es sich um ein Paket, welches Klassen und Schnittstellen zur Unterstützung von wiederverwendbaren Softwarekomponenten – die sogenannten Beans – und Möglichkeiten zur Manipulation zur Verfügung stellt.

Anwendung

Die hier enthaltenen Klassen und Schnittstellen stehen in direktem Bezug zu der Entwicklung von JavaBeans. Einige der Klassen werden von einem Bean zur Laufzeit in einer Applikation verwendet. So wird von einem Bean beispielsweise auf die Ereignisklassen zugegriffen, wenn Eigenschaften sich ändern (`PropertyChangeEvent`). Die meisten der Klassen in diesem Paket sind jedoch dafür gedacht, von einem Bean-Editor (einer Entwicklungsumgebung, um Beans zu konfigurieren und in einer Applikation zusammenzusetzen) verwendet zu werden. Mit diesen Klassen kann über den Bean-Editor eine Anwenderschnittstelle erstellt werden, über die der Anwender ein Bean nach persönlichen Vorstellungen umkonfigurieren kann.

JDK-Version

Seit 1.1

Warnungen

Die konkrete Erstellung von Beans gehört nicht mehr direkt zum JDK, sondern ist ab der aktuellen Version der Java-Plattform in ein eigenes Entwicklungspaket ausgelagert. Dieses API-Paket unterstützt dieses eigenständige Entwicklungspaket nur.

Enthaltene Klassen

```
public class BeanDescriptor extends FeatureDescriptor
public class EventSetDescriptor extends FeatureDescriptor
public class FeatureDescriptor extends Object
public class IndexedPropertyDescriptor extends PropertyDescriptor
public class Introspector extends Object
public class MethodDescriptor extends FeatureDescriptor
public class ParameterDescriptor extends FeatureDescriptor
public class PropertyChangeEvent extends EventObject
public class PropertyChangeSupport extends Object implements Serializable
public class PropertyDescriptor extends FeatureDescriptor
public class PropertyEditorManager extends Object
public class PropertyEditorSupport extends Object implements PropertyEditor
public class SimpleBeanInfo extends Object implements BeanInfo
public class VetoableChangeSupport extends Object implements Serializable
```

Enthaltene Schnittstellen

```
public abstract interface AppletInitializer
public abstract interface BeanInfo
public abstract interface Customizer
public abstract interface DesignMode
public abstract interface PropertyChangeListener extends EventListener
public abstract interface PropertyEditor
public abstract interface VetoableChangeListener extends EventListener
public abstract interface Visibility
```

Enthaltene Ausnahmen

```
public class IntrospectionException extends Exception
public class PropertyVetoException extends Exception
```

java.beans.beancontext

Die Unterstützung eines Beancontexts.

Beschreibung

Unterstützende Klassen und Schnittstellen für einen Beancontext (das Umfeld eines Beans).

Anwendung

Unterstützende Klassen und Schnittstellen für einen Beancontext bedeuten einen Container für Beans und definieren eine Ausführungsumgebung für den Bean, den der Container enthält. Es können sich verschiedene Beans in einem einzelnen Beancontext befinden und ein Beancontext kann in einem anderen Beancontext eingebunden sein. Dieses Package beinhaltet des Weiteren Events und Listenerinterfaces für den Fall, dass Beans einem Beancontext hinzugefügt oder daraus entfernt werden.

JDK-Version

Seit 1.2

Warnungen

Die konkrete Erstellung von Beans gehört nicht mehr direkt zum JDK, sondern ist ab der aktuellen Version der Java-Plattform in ein eigenes Entwicklungspaket ausgelagert. Dieses API-Paket unterstützt dieses eigenständige Entwicklungspaket nur.

Enthaltene Klassen

```
public class BeanContextChildSupport extends Object implements BeanContextChild,
BeanContextServicesListener, Serializable
public abstract class BeanContextEvent extends EventObject
public class BeanContextMembershipEvent extends BeanContextEvent
public class BeanContextServiceAvailableEvent extends BeanContextEvent
public class BeanContextServiceRevokedEvent extends BeanContextEvent
public class BeanContextServicesSupport extends BeanContextSupport implements
BeanContextServices
protected static class BeanContextServicesSupport.BCSSServiceProvider extends
Object implements Serializable
public class BeanContextSupport extends BeanContextChildSupport implements
BeanContext, Serializable, PropertyChangeListener, VetoableChangeListener
protected static final class BeanContextSupport.BCSIterator extends Object
implements Iterator
```

Enthaltene Schnittstellen

```
public abstract interface BeanContext extends BeanContextChild, Collection,
DesignMode, Visibility
public abstract interface BeanContextChild
public abstract interface BeanContextChildComponentProxy
public abstract interface BeanContextContainerProxy
public abstract interface BeanContextMembershipListener extends EventListener
public abstract interface BeanContextProxy
public abstract interface BeanContextServiceProvider
public abstract interface BeanContextServiceProviderBeanInfo extends BeanInfo
public abstract interface BeanContextServiceRevokedListener extends EventListener
public abstract interface BeanContextServices extends BeanContext,
BeanContextServicesListener
public abstract interface BeanContextServicesListener extends
BeanContextServiceRevokedListener
```

Enthaltene Ausnahmen

(keine)

java.io

Die Ein- und Ausgabe unter Java.

Beschreibung

Das Paket beinhaltet die grundlegenden Techniken zur Ein- und Ausgabe unter Java per Datenströmen, Serialization und für das Dateisystem.

Anwendung

Dieses Paket ist eines der wichtigsten des Java-API und dementsprechend umfangreich. Ein- und Ausgabeoperationen zählen zu den grundlegendsten Aktionen, welche von einem Programm bewerkstelligt werden. Ob es nur um das Abspeichern und Wiedereinlesen von Einstellungen geht oder gleich um ganze Dokumente von zig Seiten. Kaum ein Programm kommt noch ohne Ein- und Ausgabeoperationen aus.

Ein- und vor allem Ausgabe bedeuten in klassischen Anwendungen, dass Daten aus einer Datei gelesen oder in eine Datei geschrieben werden. Innerhalb von Applets ist dies meist durch die Einstellungen des Containers grundsätzlich verboten. Die Klassen zur Ein- und Ausgabe sind also in der Regel eher für echte Java-Applikationen als für Applets gedacht.

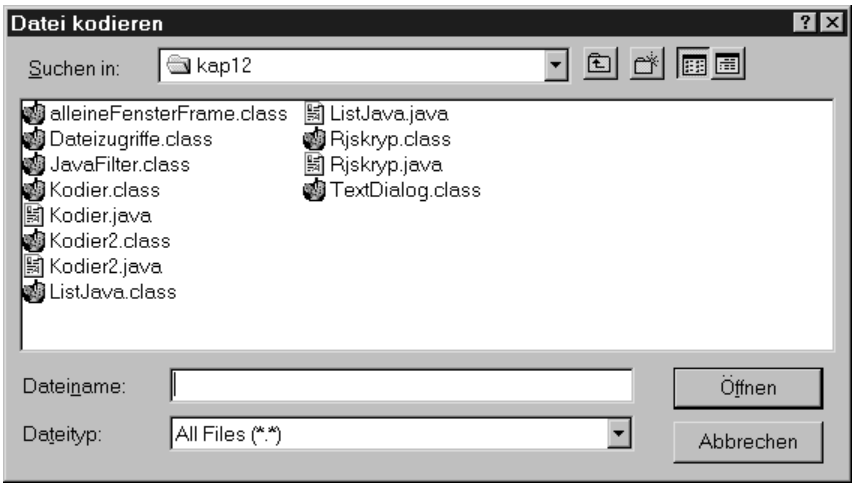
In Java werden Ein- und Ausgabeoperationen sehr oft mittels sogenannter Datenströme realisiert. Der Begriff Strom geht auf Unix zurück – das Pipe-Betriebssystem. Unter einer Pipe versteht man einen nichtinterpretierten Strom von Bytes. Er wird zur Kommunikation von Programmen untereinander beziehungsweise von Programmen und Hardwareschnittstellen verwendet.

Von entscheidender Bedeutung ist, dass ein Datenstrom von jeder beliebigen Quelle herkommen kann, das bedeutet der Ursprungsort spielt überhaupt keine Rolle. In dem Datenstrom selbst steckt die Information über die Quelle in einem Stromargument sowie ein weiteres Argument für die zu sendenden Daten.

Java stellt eine Vielzahl von unterschiedlichen Strömen bereit. Im Wesentlichen basieren sie auf den abstrakten Klassen `InputStream` und `OutputStream`, welche zu dem Paket `java.io` gehören. Die Klassen `IntputStream` und `OutputStream` bieten Zugriff auf eine Folge von Dateneinheiten, normalerweise im 8-Bit-Format. Von einem `InputStream`-Objekt kann gelesen werden. Auf ein `OutputStream`-Objekt kann geschrieben werden. Unterklassen von `InputStream` und `OutputStream` können zahlreiche Filterfunktionen implementieren, um gezielter lesen und schreiben zu können.

Es gibt darüber hinaus in dem Paket auch Klassen, die nicht auf diese beiden abstrakten Klassen zurückgehen. Eine der wohl wichtigsten Nicht-Streamklassen ist die Klasse `File`, welche Dateinamen und Verzeichnisnamen in einer plattformunabhängigen Weise verwaltet. Die Klasse `File` bietet Dienste an, um auf lokalen

Systemen Dateien und Verzeichnisse aufzulisten, zu lesen und schreiben, Dateiattribute zu erfragen und Dateinamen zu ändern oder zu löschen. Dabei greift Java auf hoher Ebene auf die von der Plattform zur Verfügung gestellten Funktionalitäten zu und nutzt diese (etwa einen Speichern-Dialog unter Windows).



Speichern von Daten in Java erfolgt bei Bedarf über Standarddialoge der Betriebssystemplattform.

Auch jenseits der konkreten `File`-Klasse gibt es Möglichkeiten, auf Datenträgern zu lesen und zu schreiben. Über die Klasse `RandomAccessFile` lassen sich beispielsweise Zeichen an willkürlichen Positionen innerhalb von Dateien auslesen. Weitere Eingabe und Ausgabe von Daten erfolgt über Unterklassen der Klassen `Reader` und `Writer`.

Zwar verfügen einige Ströme über einige spezielle Methoden, aber es gibt bestimmte Methoden, die fast immer zur Verfügung stehen. Etwa die Methoden der Familien `read()` und `write()` in zahlreichen Ausprägungen und Abarten. In einigen Klassen gibt es davon abgeleitete Methoden mit leicht erweitertem Namen wie `readByte()`, `readChar()`, `readShort()`, `writeInt(int i)`, `writeLong(long l)`, `writeFloat(float f)` oder `writeDouble(double d)`.

Die einfachen Strom-Methoden erlauben nur das Versenden von Bytes als Datenstrom. Zum Senden verschiedener Datentypen gibt es die Schnittstellen `DataInput` und `DataOutput`. Sie legen Methoden zum Senden und Empfangen anderer Java-Datentypen fest. Mit Hilfe der Schnittstellen `ObjectInput` und `ObjectOutput` lassen sich ganze Objekte über einen Strom senden. Mit dem `StreamTokenizer` kann ein Strom wie eine Gruppe von Worten behandelt werden. Eine weitere Streamklasse – `StringBufferInputStream` – liest Daten aus dem StringBuffer und ermöglicht damit eine Cache-Funktionalität. Für die Kommunikation von einzelnen Threads gibt es die beiden Klassen `PipedInputStream` zum Lesen von Daten aus einem `PipedOutputStream`.

Im Zusammenhang mit der verteilten Programmierung unter Java (RMI) wurde das Konzept der sogenannten Object Serialization aufgenommen. Es ermöglicht das Abspeichern der Inhalte eines Objektes in einen Stream. Dies kann aber auch im Prinzip eine beliebige Datei sein. Ein Objekt kann in einem Stream zwischengespeichert und zu einem späteren Zeitpunkt daraus wieder aufgebaut werden. Die Lebensdauer eines Objektes wird in diesem Konzept also über die eigentliche Laufzeit eines Programms hinaus verlängert. Hauptanwendung hierfür ist das Versenden von Objekten über das Netzwerk im Zusammenhang mit RMI. Das Java-Package `java.io` beinhaltet seit dem JDK 1.2 die entsprechenden Erweiterungen dafür und ergänzt damit die Haupt-RMI-Pakete (`java.rmi`, `java.rmi.dgc`, `java.rmi.registry` und `java.rmi.server`). `serialver` heisst das Object-Serialization-Tool für Java unter dem JDK.

Alle Methoden, welche sich mit Eingabe- und Ausgabeoperationen beschäftigen, werden in der Regel mit `throw IOException` abgesichert. Diese Subklasse von `Exception` enthält alle potentiellen I/O-Fehler, welche bei der Verwendung von Datenströmen auftreten können. Daneben gibt es in dem Paket zahlreiche weitere Ausnahmen, um auf jede Ausnahmesituation qualifiziert reagieren zu können und die meist eine direkte Erweiterung von `Exception` sind.

JDK-Version

Seit 1.0

Warnungen

Die nachfolgend noch angegebenen Klassen `LineNumberInputStream` und `StringBufferInputStream` gelten mittlerweile als deprecated.

Enthaltene Klassen

```
public class BufferedInputStream extends FilterInputStream
public class BufferedOutputStream extends FilterOutputStream
public class BufferedReader extends Reader
public class BufferedWriter extends Writer
public class ByteArrayInputStream extends InputStream
public class ByteArrayOutputStream extends OutputStream
public class CharArrayReader extends Reader
public class CharArrayWriter extends Writer
public class DataInputStream extends FilterInputStream implements DataInput
public class DataOutputStream extends FilterOutputStream implements DataOutput
public class  extends Object implements Serializable, Comparable
public final class FileDescriptor extends Object
public class FileInputStream extends InputStream
public class FileOutputStream extends OutputStream
public final class FilePermission extends Permission implements Serializable
public class FileReader extends InputStreamReader
public class FileWriter extends OutputStreamWriter
```

```
public class FilterInputStream extends InputStream
public class FilterOutputStream extends OutputStream
public abstract class FilterReader extends Reader
public abstract class FilterWriter extends Writer
public abstract class InputStream extends Object
public class InputStreamReader extends Reader
public class LineNumberInputStream extends FilterInputStream
public class LineNumberReader extends BufferedReader
public class ObjectInputStream extends InputStream implements ObjectInput,
ObjectStreamConstants
public abstract static class ObjectInputStrea.GetField extends Object
public class ObjectOutputStream extends OutputStream implements ObjectOutput,
ObjectStreamConstants
public abstract static class ObjectOutputStream.PutField extends Object
public class ObjectStreamClass extends Object implements Serializable
public class ObjectStreamField extends Object implements Comparable
public abstract class OutputStream extends Object
public class OutputStreamWriter extends Writer
public class PipedInputStream extends InputStream
public class PipedOutputStream extends OutputStream
public class PipedReader extends Reader
public class PipedWriter extends Writer
public class PrintStream extends FilterOutputStream
public class PrintWriter extends Writer
public class PushbackInputStream extends FilterInputStream
public class PushbackReader extends FilterReader
public class RandomAccessFile extends Object implements DataOutput, DataInput
public abstract class Reader extends Object
public class SequenceInputStream extends InputStream
public final class SerializablePermission extends BasicPermission
public class StreamTokenizer extends Object
public class StringBufferInputStream extends InputStream
public class StringReader extends Reader
public class StringWriter extends Writer
public abstract class Writer extends Object
```

Enthaltene Schnittstellen

```
public abstract interface DataInput
public abstract interface DataOutput
public abstract interface Externalizable extends Serializable
public abstract interface FileFilter
public abstract interface FilenameFilter
public abstract interface ObjectInput extends DataInput
public abstract interface ObjectInputValidation
public abstract interface ObjectOutput extends DataOutput
public abstract interface ObjectStreamConstants
public abstract interface Serializable
```

Enthaltene Ausnahmen

```
public class CharConversionException extends IOException
public class EOFException extends IOException
public class FileNotFoundException extends IOException
public class InterruptedIOException extends IOException
public class InvalidClassException extends ObjectStreamException
public class InvalidObjectException extends ObjectStreamException
public class IOException extends Exception
public class NotActiveException extends ObjectStreamException
public class NotSerializableException extends ObjectStreamException
public abstract class ObjectStreamException extends IOException
public class OptionalDataException extends ObjectStreamException
public class StreamCorruptedException extends ObjectStreamException
public class SyncFailedException extends IOException
public class UnsupportedEncodingException extends IOException
public class UTFDataFormatException extends IOException
public class WriteAbortedException extends ObjectStreamException
```

java.lang

Die fundamentalen Klassen des Java-Designs.

Beschreibung

Das wohl wichtigste Paket von Java, denn dieses Paket beinhaltet den Kern der Java-Sprache.

Anwendung

Das Package beinhaltet den Kern der Java-Sprache und dürfte somit das wichtigste Paket sein, um überhaupt mit Java arbeiten zu können. Das Paket wird immer automatisch in alle Java-Klassen eingebunden. Implizit besitzen also alle Java-Quelltexte die `import`-Anweisung für das `java.lang`-Paket. Es enthält alle wesentlichen Java-Klassen.

Die Klasse `Object` ist hier enthalten, welche die Wurzel der gesamten Klassenhierarchie ist. Aber auch zahlreiche andere für die Java-Sprache extrem wichtigen Grundklassen sind in dem Paket integriert.

So beispielsweise die Klasse `Class`. Instanzen davon repräsentieren Java-Klassen und -Schnittstellen in einer laufenden Applikation. Die Klasse `System` wird in fast jeder Applikation Anwendung finden. Die darin deklarierten Variablen `static PrintStream err`, der Standard-Error-Ausgabestrom, `static InputStream in`, der Standard-Eingabestrom, und `static PrintStream out`, der Standard-Ausgabestrom von Java, sind wichtige Details. Gerade die Variable `out` findet permanent Anwendung in der klassischen Ausgabemethode `System.out.println()`.

Aber auch ganz elementare Java-Methoden sind in dieser Klasse vorhanden, etwa `static void exit(int status)` zum Beenden der JVM oder `static void gc()` zum direkten Aufruf des Garbage Collectors.

`java.lang` ist desgleichen Grundlage des gesamten Ausnahme- und Fehlerbehandlungskonzeptes von Java. Die hier enthaltene Klasse `Throwable` ist die Superklasse von allen Errors und Exceptions in der gesamten Java-Sprache. Nur diejenigen Objekte, welche Instanz dieser Klasse oder einer ihrer Subklassen sind, können in Java durch die JVM über ein `throw`-Statement ausgeworfen werden und ein Argumententyp in einer `catch`-Klausel sein. Diese Klasse ergänzend sind in dem Paket `java.lang` die darauf aufgebauten, grundlegenden Ausnahme- und Fehlerklassen der Javasprache enthalten. Etwa `SecurityException` für Sicherheitsverletzungen, `ArithmeticException` für Ausnahmen bei arithmetischen Operationen, `ArrayIndexOutOfBoundsException` für Zugriffe auf Elemente eines Arrays, die nicht definiert sind, oder `ClassNotFoundException`, wenn eine Klasse bei einem Ladevorgang nicht gefunden wird. Insbesondere befindet sich in dem Paket jedoch die Ausnahmeklasse `Exception`, welche für zahlreiche Ausnahmen direkte oder indirekte

Superklasse ist, und `RuntimeException`, welche die Superklasse von all den Ausnahmen ist, die durch normale Operationen während der Laufzeit der JVM ausgeworfen werden.

Das Package stellt darüber hinaus zahlreiche Fehlerklassen zur Verfügung, um auf die wesentlichen Fehler in Java reagieren zu können. Insbesondere die Fehlerklasse `Error` zum Behandeln von all den ernsthaften Problemen, welche nicht durch eine Applikation per `try-catch` abgefangen werden sollten, aber auch `InternalError` zum Behandeln von internen Problemen der JVM.

Ganz wichtig im `java.lang`-Paket ist gleichermaßen die Klasse `ClassLoader`, welche für den Ladevorgang von Klassen von fundamentaler Bedeutung ist, oder die Klasse `SecurityManager`, auf der das gesamte Sicherheitskonzept von Java aufbaut.

Ebenso befinden sich Klassen für sämtliche primitive Typen in dem Paket (sogenannte Wrapper-Klassen). Die Klassen `Boolean`, `Character`, `Integer`, `Long`, `Float` und `Double` repräsentieren die jeweiligen primitiven Typen und erlauben die Erzeugung von Typobjekten. Diese werden beispielsweise gebraucht, wenn Referenzen auf primitive Werte benötigt werden. Objekte dieser Klassen repräsentieren dann die Werte der entsprechenden primitiven Typen (eine Auswahl):

- Ein `Boolean`-Objekt repräsentiert einen `boolean`-Wert.
- Ein `Byte`-Objekt steht für einen `byte`-Wert.
- Ein `Character`-Objekt repräsentiert einen `char`-Wert.
- Ein `Integer`-Objekt repräsentiert einen `int`-Wert.
- Ein `Long`-Objekt repräsentiert einen `long`-Wert.
- Ein `Float`-Objekt repräsentiert einen `float`-Wert.
- Ein `Double`-Objekt repräsentiert einen `double`-Wert.

Die Konstruktoren dieser Klassen sind so definiert, dass sich Objekte dieser Klassen mit primitiven Werten des entsprechenden Typs oder mit `String`-Objekten initialisieren lassen. `String`-Objekte werden dabei in die entsprechenden primitiven Werte umgewandelt. Die Klassen beinhalten unter anderem Methoden, die den primitiven Wert, den sie repräsentieren, zurückgeben, und Methoden, die einen String in den entsprechenden primitiven Wert umwandeln.

Auch wichtige Schnittstellen sind in dem Paket vorhanden. Etwa die Schnittstelle `Runnable`, welche im Rahmen des Multithreading-Konzeptes von zentraler Bedeutung ist (zusammen mit den ebenfalls in dem Paket zu findenden Klassen `Thread`, `ThreadGroup` und `ThreadLocal`).

JDK-Version

Seit 1.0

Enthaltene Klassen

```
public final class Boolean extends Object implements Serializable
public final class Byte extends Number implements Comparable
public final class Character extends Object implements Serializable, Comparable
public static class Character.Subset extends Object
public static final class Character.UnicodeBlock extends Character.Subset
public final class Class extends Object implements Serializable
public abstract class ClassLoader extends Object
public final class Compiler extends Object
public final class Double extends Number implements Comparable
public final class Float extends Number implements Comparable
public class InheritableThreadLocal extends ThreadLocal
public final class Integer extends Number implements Comparable
public final class Long extends Number implements Comparable
public final class Math extends Object
public abstract class Number extends Object implements Serializable
public class Object
public class Package extends Object
public abstract class Process extends Object
public class Runtime extends Object
public final class RuntimePermission extends BasicPermission
public class SecurityManager extends Object
public final class Short extends Number implements Comparable
public final class String extends Object implements Serializable, Comparable
public final class StringBuffer extends Object implements Serializable
public final class System extends Object
public class Thread extends Object implements Runnable
public class ThreadGroup extends Object
public class ThreadLocal extends Object
public class Throwable extends Object implements Serializable
public final class Void extends Object
```

Enthaltene Schnittstellen

```
public abstract interface Cloneable
public abstract interface Comparable
public abstract interface Runnable
```

Enthaltene Ausnahmen

```
public class ArithmeticException extends RuntimeException
public class ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException
public class ArrayStoreException extends RuntimeException
public class ClassCastException extends RuntimeException
public class ClassNotFoundException extends Exception
public class CloneNotSupportedException extends Exception
public class Exception extends Throwable
public class IllegalAccessException extends Exception
public class IllegalArgumentException extends RuntimeException
public class IllegalMonitorStateException extends RuntimeException
```

```
public class IllegalStateException extends RuntimeException
public class IllegalThreadStateException extends IllegalArgumentException
public class IndexOutOfBoundsException extends RuntimeException
public class InstantiationException extends Exception
public class InterruptedException extends Exception
public class NegativeArraySizeException extends RuntimeException
public class NoSuchFieldException extends Exception
public class NoSuchMethodException extends Exception
public class NullPointerException extends RuntimeException
public class NumberFormatException extends IllegalArgumentException
public class RuntimeException extends Exception
public class SecurityException extends RuntimeException
public class StringIndexOutOfBoundsException extends IndexOutOfBoundsException
public class UnsupportedOperationException extends RuntimeException
```

Enthaltene Errors

```
public class AbstractMethodError extends IncompatibleClassChangeError
public class ClassCircularityError extends LinkageError
public class ClassFormatError extends LinkageError
public class Error extends Throwable
public class ExceptionInInitializerError extends LinkageError
public class IllegalAccessError extends IncompatibleClassChangeError
public class IncompatibleClassChangeError extends LinkageError
public class InstantiationError extends IncompatibleClassChangeError
public class InternalError extends VirtualMachineError
public class LinkageError extends Error
public class NoClassDefFoundError extends LinkageError
public class NoSuchFieldError extends IncompatibleClassChangeError
public class NoSuchMethodError extends IncompatibleClassChangeError
public class OutOfMemoryError extends VirtualMachineError
public class StackOverflowError extends VirtualMachineError
public class ThreadDeath extends Error
public class UnknownError extends VirtualMachineError
public class UnsatisfiedLinkError extends LinkageError
public class UnsupportedClassVersionError extends ClassFormatError
public class VerifyError extends LinkageError
public abstract class VirtualMachineError extends Error
```

java.lang.ref

Die Unterstützung von Objektreferenzen.

Beschreibung

Hierbei handelt es sich um ein Paket, um Objektreferenzen wie jedes andere Objekt zu behandeln.

Anwendung

Das Paket stellt `Reference-Object`-Klassen bereit, welche einen begrenzten Grad von Interaktion mit dem Garbage Collector, dem Speicherbereinigungsmechanismus von Java, erlauben. Ein Programn kann eine Objektreferenz nutzen, um eine Referenz auf ein anderes Objekt aufrecht zu erhalten in dem Sinn, dass sich das Objekt darüber immer noch durch den Garbage Collector im Zugriff befindet. Dieses neu im JDK 1.2 eingeführte `Reference-Object`-Konzept dient dazu, dass eine Referenz auf ein Objekt wie ein Objekt selbst behandelt wird und damit genauso verwendet werden kann. Dies funktioniert sogar noch dann, wenn auf das Objekt selbst gar nicht mehr zugegriffen werden kann (etwa, weil der Garbage Collector das Objekt gelöscht hat). Eine potentielle Anwendung für diese Referenzobjekte ist der Aufbau eines einfachen Caches.

JDK-Version

Seit 1.2

Enthaltene Klassen

```
public class PhantomReference extends Reference
public abstract class Reference extends Object
public class ReferenceQueue extends Object
public class SoftReference extends Reference
public class WeakReference extends Reference
```

Enthaltene Schnittstellen

(keine)

Enthaltene Ausnahmen

(keine)

java.lang.reflect

Die Unterstützung von sogenannter Reflektion.

Beschreibung

Eine Schnittstelle und einige Klassen zur Informationsbeschaffung über geladene Klassen und Objekte.

Anwendung

Hierbei handelt es sich um ein Paket zur Unterstützung von dem Konzept der Reflection, das bedeutet Informationen über geladene Klassen zu bekommen (beispielsweise ihre Attribute). Die Klassen und das Interface in dem Paket beinhalten reflektive Informationen über Klassen und Objekte. Dies erlaubt programmtechnischen Zugriff auf Variablen, Methoden und Konstruktoren von geladenen Klassen und die Verwendung von den reflektierten Variablen, Methoden und Konstruktoren.

Über die Methoden der Schnittstelle `Member` kann man beispielsweise Informationen über ein bestimmtes Mitglied einer Klasse/Objektes oder Schnittstelle herausfinden. Die Klassen selbst erlauben den spezifischen Zugriff auf bestimmte Elemente, etwa über die Klasse `Array` auf Felder, `Field` auf Variablen, `Method` auf Methoden oder `Modifier` auf Zugriffsmodifier.

JDK-Version

Seit 1.1

Enthaltene Klassen

```
public class AccessibleObject extends Object
public final class Array extends Object
public final class Constructor extends AccessibleObject implements Member
public final class Field extends AccessibleObject implements Member
public final class Method extends AccessibleObject implements Member
public class Modifier extends Object
public final class ReflectPermission extends BasicPermission
```

Enthaltene Schnittstellen

```
public abstract interface Member
```

Enthaltene Ausnahmen

```
public class InvocationTargetException extends Exception
```

java.math

Ein Paket mit Klassen für große Zahlen.

Beschreibung

Das Paket stellt zwei Klassen bereit, welche die Behandlung von großen Zahlen unterstützen.

Anwendung

Die beiden Klassen `BigInteger` und `BigDecimal` sind das Analogon zu den jeweiligen primitiven Java-Typen. Zusätzlich zu den normalen arithmetischen Operationen stellen die beiden Klassen (teilweise nur für Ganzzahlen, teils nur für Dezimalzahlen – je nachdem ob sinnvoll) modulare Arithmetik, GCD-Berechnung, Primzahlgeneration, Bit-Manipulation, ausgedehnte Kontrolle der Rundungsvorgänge und zahlreiche andere erweiternde Operationen zur Verfügung.

JDK-Version

Seit 1.1

Enthaltene Klassen

```
public class BigDecimal extends Number implements Comparable
public class BigInteger extends Number implements Comparable
```

Enthaltene Schnittstellen

(keine)

Enthaltene Ausnahmen

(keine)

java.net

Die zentralen Klassen für die Netzwerkfunktionalität von Java.

Beschreibung

Das Paket beinhaltet die wesentlichen Aspekte der Netzwerkfähigkeit von Java.

Anwendung

Durch seine Internet-Orientierung ist es für Java zwingend, Netzwerkfähigkeiten mitzubringen. Es muss also möglich sein, zwischen einem Applet oder einer Java-Applikation über ein Computernetzwerk eine Verbindung zu einem anderen System aufzubauen. Java stellt über das Paket `java.net` die wichtigsten Klassen und Schnittstellen für die einfache Realisierung von Vernetzungsoperationen zur Verfügung. So finden sich in den Schnittstellen `ContentHandlerFactory` Methoden zum Erstellen von `ContentHandler`-Objekten, `FileNameMap` Methoden zum Mappen zwischen Dateinamen und MIME-Typen, `SocketImplFactory` Methoden zum Erstellen von `Socket`-Implementationen, `URLStreamHandlerFactory` Methoden zum Erstellen von `URLStreamHandler`-Objekten und in `SocketOptions` `Socket`-Optionen.

Man unterscheidet in Java – wie auch sonst üblich – im Wesentlichen zwischen paketorientierter Kommunikation über das Netzwerk und der verbindungsorientierten Kommunikation.

Bei der ersten Variante – der paketorientierten Verbindungsart – versendet der Absender einfach ein Datenpaket mit der Adresse des Zielrechners (etwa IP-Adresse und Port bei auf TCP/IP aufsetzender Verbindung) und kümmert sich nicht weiter darum (weder um den genauen Weg, noch darum, ob es überhaupt ankommt). Die Weiterleitung obliegt dann dem Netz selbst (etwa in Form von Routern).

Die zweite Variante baut eine permanente Verbindung zu einem Zielrechner auf, die im Laufe der Kommunikation immer wieder verwendet wird. Sie wird einmal aufgebaut und dann gehalten, bis die Arbeit abgeschlossen ist – eine semi-permanente Verbindung beziehungsweise eine verbindungsorientierte Kommunikation.

Die Java-Klassen stellen äußert umfangreiche Funktionalitäten für beide Verbindungsarten bereit. Etwa über `Authenticator` Techniken zur Beglaubigung in Netzwerken. Ergänzend stellen `NetPermission` Eigenschaften zur Bereitstellung von Netzzugriffskontrolle und `PasswordAuthentication` Username und Passwort für die Verwendung bei `Authenticator` bereit.

Über `ContentHandler` haben Anwender abstrakte Eigenschaft zum Lesen von Daten in einer URL-Verbindung und Konstruieren des entsprechenden lokalen Objekts aufgrundlage des MIME-Typs zur Verfügung. Die Klassen `DatagramPacket` und `DatagramSocket` unterstützen `Datagram`-Pakete (UDP) und `Datagram`-Sockets, `DatagramSocketImpl` dient als abstrakte Basisklasse für `Datagram`- und `Multicast`-Sockets.

`HttpURLConnection` erlaubt dagegen eine Verbindung mittels dem HTTP-Protokoll, `InetAddress` eine Objektrepräsentation von einem Internethost (Hostname, IP-Addresse).

`JarURLConnection` repräsentiert eine URL-Verbindung zu einer JAR-Datei, während URL eine Objektrepräsentation von einem beliebigen URL darstellt. Die Klassen `URLClassLoader` (URL-Suchpfad), `URLConnection` (abstrakte Eigenschaft für einen Socket, welcher verschiedene Web-basierende Protokolle (http, ftp, usw.) handhaben kann), `URLDecoder` (Konvertierung von MIME-Zeichen – `x-www-form-urlencoded`-Format – in Strings), `URLEncoder` (Verwandlung von Strings in das `x-www-form-urlencoded`-Format) und `URLStreamHandler` (abstrakte Klasse zum Handhaben von Objektströmen, auf die URLs verweisen) stellen sämtliche Funktionen zur Verfügung, welche bei URL-basierender Verbindung notwendig ist.

Für Socket-basierende Kommunikation stellt Java mit den Klassen `MulticastSocket` (ein serverseitiges Socket mit Unterstützung für übermittelte Daten an mehre Client-Sockets), `ServerSocket` (serverseitiges Socket), `Socket` (ein Client-seitiges Socket), `SocketImpl` (abstrakte Klasse für die spezifische Socket-Implementation), `SocketPermission` (Eigenschaften zur Bereitstellung von Netzzugriffen via Sockets) zahlreiche Möglichkeiten zur Verfügung.

Die umfangreiche Liste an Ausnahmen erlaubt darüber hinaus die qualifizierte Reaktion auf die meisten nichtplanbaren Standardsituationen bei Netzwerkverbindungen.

Dies alles versetzt Java in die Lage, relativ einfach plattformübergreifende Netzwerkapplikationen zu realisieren. Darunter fallen das Lesen und Schreiben von Dateien über das Netzwerk oder das Verbinden mit den üblichen Internet-Protokollen. Zwar gibt es aus Sicherheitsgründen erhebliche Einschränkungen von Lese- und Schreiboperationen bei Applets, aber ansonsten sind Netzwerkzugriffe mit Java leichter zu realisieren als mit den meisten anderen Sprachen.

JDK-Version

Seit 1.0

Enthaltene Klassen

```
public abstract class Authenticator extends Object
public abstract class ContentHandler extends Object
public final class DatagramPacket extends Object
public class DatagramSocket extends Object
public abstract class DatagramSocketImpl extends Object implements SocketOptions
public abstract class HttpURLConnection extends URLConnection
public final class InetAddress extends Object implements Serializable
public abstract class JarURLConnection extends URLConnection
public class MulticastSocket extends DatagramSocket
public final class NetPermission extends BasicPermission
```

```
public final class PasswordAuthentication extends Object
public class ServerSocket extends Object
public class Socket extends Object
public abstract class SocketImpl extends Object implements SocketOptions
public final class SocketPermission extends Permission implements Serializable
public final class URL extends Object implements Serializable
public class URLClassLoader extends SecureClassLoader
public abstract class URLConnection extends Object
public class URLDecoder extends Object
public class URLEncoder extends Object
public abstract class URLStreamHandler extends Object
```

Enthaltene Schnittstellen

```
public abstract interface ContentHandlerFactory
public abstract interface FileNameMap
public abstract interface SocketImplFactory
public abstract interface SocketOptions
public abstract interface URLStreamHandlerFactory
```

Enthaltene Ausnahmen

```
public class BindException extends SocketException
public class ConnectException extends SocketException
public class MalformedURLException extends IOException
public class NoRouteToHostException extends SocketException
public class ProtocolException extends IOException
public class SocketException extends IOException
public class UnknownHostException extends IOException
public class UnknownServiceException extends IOException
```

java.rmi

Die grundlegende Unterstützung von verteilten Java-to-Java-Applikationen.

Beschreibung

Das Paket ist ein zentraler Bestandteil des API für das RMI-Modell von Java. Das Remote Method Invocation Interfaces (RMI) bietet die Möglichkeit, Java-Klassen, welche auf einer anderen virtuellen Maschine laufen, anzusprechen. Dabei ist es egal, ob die virtuelle Maschine lokal vorhanden oder irgendwo in einem Netzwerk ausgeführt wird.

Anwendung

RMI stellt mit diesem Paket einen wesentlichen Bestandteil des API zur Verfügung, mit dessen Hilfe eine Kommunikation zweier Java-Komponenten über Adress- oder Maschinenräume hinweg möglich ist. RMI ist eine Art objektorientierter RPC-Mechanismus (Remote Procedure Call), der speziell für Java entwickelt wurde. Das heisst, dass RMI nur genutzt werden kann, wenn sowohl das Server- als auch das Client-Objekt in Java implementiert sind. Deswegen wird auch keine spezielle Beschreibungssprache benutzt, um das entfernte Interface zu beschreiben. Über die Schnittstelle `Remote` werden Methoden zum Identifizieren entfernter (Remote-) Objekte deklariert, während die Klassen unter anderem Stromrepräsentationen von Objekten (`MarshalledObject`), Eigenschaften zum Behandeln von Referenzen auf Remote-Objekte – basierend auf der URL-Syntax (`Naming`) – oder Methoden zur Definition einer RMI Stub Security Policy für Applikationen (`RMISecurityManager`) bereitstellen.

Auffällig an `java.rmi` ist, dass das Paket – insbesondere im Vergleich zu der Anzahl der enthaltenen Schnittstellen und Klassen – eine ungewöhnlich umfangreiche Anzahl von vordefinierten Ausnahmen bereitstellt.

In Java werden zur vollständigen Umsetzung des RMI-Konzeptes neben dem `java.rmi`-Paket noch die drei Packages `java.rmi.dgc`, `java.rmi.registry` und `java.rmi.server` verwendet. Das JDK stellt mit `rmic` – dem Java RMI Stub Compiler –, `rmid` – Java RMI Activation System Daemon – und `rmiregistry` – Java Remote Object Registry – drei Tools zur programmiertechnischen Umsetzung des RMI-Konzeptes bereit.

Im Zusammenhang mit dem Konzept der Object Serialization (das Abspeichern der Inhalte eines Objektes in einen Stream – etwa in einer beliebigen Datei) und dem Versenden von Objekten über ein Netzwerk erweitert das Standardpackage zur Ein- und Ausgabe (`java.io`) die Funktionalität von RMI.

Das RMI-Konzept ist nicht nur auf Java-Plattformen beschränkt, sondern kann über die Java-IDL (Interfaces Definition Language) eine Verbindung zu anderen Verteilungsplattformen, wie beispielsweise CORBA (Common Object Request

Broker Architecture), aufbauen. RMI und CORBA werden über IIOP (Internet InterORB Protocol) zusammengebracht. Das neue JDK enthält einen ORB (Object Request Broker), welcher es erlaubt, verteilte Anwendungen auf der Basis von Java und CORBA zu schreiben. Während RMI eine reine Java-Lösung darstellt, ist CORBA eine von Java vollkommen losgelöste Lösung für verteilte Strukturen, die mit beliebigen Sprachen eingesetzt werden. IDL dient der Definition von CORBA-Komponenten in Java.

JDK-Version

Seit 1.1

Warnungen

Die nachfolgend noch angegebenen Ausnahmen `RMISecurityException` und `ServerRuntimeException` gelten mittlerweile als deprecated. Für sie gibt es auch keinen unmittelbaren Ersatz, denn im RMI-Konzept der Java-2-Plattform werden deren Funktionalitäten nicht mehr benötigt (die Form der Ausnahmen werden im RMI-2- Konzept nicht mehr ausgeworfen – wohl aber, wenn nach dem Vorläufer-RMI-Konzept programmiert wird).

Enthaltene Klassen

```
public final class MarshalledObject extends Object implements Serializable
public final class Naming extends Object
public class RMISecurityManager extends SecurityManager
```

Enthaltene Schnittstellen

```
public abstract interface Remote
```

Enthaltene Ausnahmen

```
public class AccessException extends RemoteException
public class AlreadyBoundException extends Exception
public class ConnectException extends RemoteException
public class ConnectIOException extends RemoteException
public class MarshalException extends RemoteException
public class NoSuchObjectException extends RemoteException
public class NotBoundException extends Exception
public class RemoteException extends IOException
public class RMISecurityException extends SecurityException
public class ServerError extends RemoteException
public class ServerException extends RemoteException
public class ServerRuntimeException extends RemoteException
public class StubNotFoundException extends RemoteException
public class UnexpectedException extends RemoteException
public class UnknownHostException extends RemoteException
public class UnmarshalException extends RemoteException
```

java.rmi.activation

Ergänzende Funktionalität für das Java-Konzept der verteilten Programmierung.

Beschreibung

Das Paket stellt Techniken für das Konzept der RMI Object Activation bereit. Eine Objektreferenz kann über den RMI-Aktivierungsmechanismus persistent gemacht und später wieder reaktiviert werden, damit daraus wieder ein »lebendes« Objekt wird.

Anwendung

Das neu im JDK 1.2 eingeführte Paket unterstützt persistente (beständige) Referenzen auf Remote-Objekte und die automatische Objekt-Reaktivierung über diese Referenzen. Dazu gibt es Schnittstellen wie `ActivationInstantiator` mit Methoden zum Erstellen von Gruppen von Objekten, die aktiviert werden können, `ActivationMonitor` zum Reagieren auf Veränderungen, `ActivationSystem` zum Registrieren von Gruppen und aktivierbaren Objekten und `Activator` mit Methoden in einem aktivierten Remote-Objekt.

Klassen wie `Activatable` repräsentieren Remote-Objekte, die über die Zeit ihres Zugriffs persistent gemacht und wieder aktiviert werden können, `ActivationGroupID` und `ActivationID` beinhalten Identifier für eine registrierte Gruppe beziehungsweise ein Objekt.

`ActivationDesc`, `ActivationGroupDesc`, `ActivationGroupDesc.CommandEnvironment` oder `ActivationGroup` stellen einen Deskriptor mit Informationen und allgemeine Informationen bereit, welche in aktivierten Objekten beziehungsweise Gruppen von solchen Objekten benötigt werden.

JDK-Version

Seit 1.2

Enthaltene Klassen

```
public abstract class Activatable extends RemoteServer
public final class ActivationDesc extends Object implements Serializable
public abstract class ActivationGroup extends UnicastRemoteObject implements
ActivationInstantiator
public final class ActivationGroupDesc extends Object implements Serializable
public static class ActivationGroupDesc.CommandEnvironment extends Object
implements Serializable
public class ActivationGroupID extends Object implements Serializable
public class ActivationID extends Object implements Serializable
```

Enthaltene Schnittstellen

```
public abstract interface ActivationInstantiator extends Remote
public abstract interface ActivationMonitor extends Remote
public abstract interface ActivationSystem extends Remote
public abstract interface Activator extends Remote
```

Enthaltene Ausnahmen

```
public class ActivateFailedException extends RemoteException
public class ActivationException extends Exception
public class UnknownGroupException extends ActivationException
public class UnknownObjectException extends ActivationException
```

java.rmi.dgc

Ergänzende Funktionalität für das Java-Konzept der verteilten Programmierung.

Beschreibung

Unterstützende Klassen und Interfaces für die RMI Distributed Garbage-Collection (DGC).

Anwendung

Dieses Paket unterstützt den Distributed Garbage Collection-Algorithmus im Rahmen des RMI-Konzeptes. Wenn ein RMI-Server ein Objekt an seinen Client (den Aufrufer von der `remote`-Methode) zurückgibt, muss das aufrufende Objekt im Client bestimmt werden. Wenn im Client keine Referenz auf das aufrufende Objekt mehr vorhanden ist, kann dies zu Schwierigkeiten führen. Unter Umständen muss sich der Server um den Aufbau der Verbindung zu dem Objekt selbst kümmern. Im Wesentlichen geht es also darum, Verbindungen zwischen Server- und Client-Objekten aufrecht zu erhalten, bei Bedarf neu aufzubauen, und gegebenenfalls endgültige Speicherbereinigungen von nicht mehr benötigten Verbindungen durchzuführen. Die Schnittstelle `DGC` stellt dazu Methoden zum Bereinigen von Verbindungen für nicht mehr verwendete Clients bereit, die Klasse `Lease` beinhaltet einen einzigartigen VM-Identifier und eine Verwendungsdauer dafür, um damit eine Verbindung zwischen Server und Remote-Objekt in jedem Fall wiederherstellen zu können. Über die `VMID`-Klasse werden Eigenschaften zur Aufrechterhaltung der einzigartigen VM-ID über alle JVM hinweg zur Verfügung gestellt.

JDK-Version

Seit 1.1

Enthaltene Klassen

```
public final class Lease extends Object implements Serializable
public final class VMID extends Object implements Serializable
```

Enthaltene Schnittstellen

```
public abstract interface DGC extends Remote
```

Enthaltene Ausnahmen

(keine)

java.rmi.registry

Ergänzende Funktionalität für das Java-Konzept der verteilten Programmierung.

Beschreibung

Bereitstellung von einer Klasse und zwei Schnittstellen für die RMI-Registry.

Anwendung

Unter einer RMI-Registry ist ein Remote-Objekt zu verstehen, welches Namen auf Remote-Objekte mappt. Ein Server registriert seine entfernten Objekte mit dieser Registry, so dass sie verschlossen werden können. Wenn ein Objekt eine Methode eines Remote-Objektes aufrufen möchte, muss dieses zuerst über Aufruf des Namens aufgeschlossen werden. Die RMI-Registry gibt zu dem aufrufenden Objekt eine Referenz auf das Remote-Objekt zurück, über die die Methode dann genutzt werden kann. Die Schnittstelle `Registry` beinhaltet Methoden zur Bereitstellung der Registratur für verschiedene Hosts, während die Klasse `LocateRegistryEigenschaften` Methoden zur Startroutineregistrierung in einem Host bereitstellt.

JDK-Version

Seit 1.1

Warnungen

Die Schnittstelle `RegistryHandler` gilt seit Java 2 als verworfen.

Enthaltene Klassen

```
public final class LocateRegistry extends Object
```

Enthaltene Schnittstellen

```
public abstract interface Registry extends Remote
public abstract interface RegistryHandler
```

Enthaltene Ausnahmen

(keine)

java.rmi.server

Ergänzende Funktionalität für das Java-Konzept der verteilten Programmierung.

Beschreibung

Hierbei handelt es sich um ein Paket für den Server-seitigen Aufruf von entfernten Methoden via RMI.

Anwendung

Die unterstützenden Klassen und Schnittstellen für den Server-seitigen Aufruf von entfernten Methoden via RMI in dem Paket teilen sich in eine Gruppe von Funktionalitäten, welche von Stubs und Skeletons verwendet werden, die mit dem `rmic`-Stubcompiler des JDK generiert werden. Eine andere Gruppe von Klassen/Schnittstellen implementiert das RMI Transport Protocol und HTTP-Tunneling.

So stellt die Schnittstelle `RMIFailureHandler` Methoden für das Handling bereit, wenn die RMI-Runtime nicht zum Erstellen eines Sockets oder ServerSockets in der Lage ist.

Die Klasse `RemoteServer` ist die Superklasse einer Serverimplementation und `RemoteStub` bietet allgemeine Unterstützung für Stub-Objekte. Über `UID` kann eine Abstraktion für das Generieren von eindeutigen Identifiern erfolgen und `UnicastRemoteObject` definiert ein nichtreplizierbares Remote-Objekt, dessen Referenz nur solange gültig ist, wie der Serverprozess läuft.

Die Klasse `ObjID` dient zur eindeutigen Identifizierung des entfernten Objektes in einer VM und `RMIClassLoader` enthält Methoden für das Laden von Klassen über ein Netzwerk. `RMISocketFactory` wird von dem RMI-Runtime zur Bereitstellung von Client- und Server-Sockets für RMI-Aufrufe verwendet und `RemoteObject` enthält die Semantik von dem entfernten Objekt.

JDK-Version

Seit 1.1

Warnungen

In diesem Paket sind seit Java 2 zahlreiche Schnittstellen und Klassen, aber auch Ausnahmen als verworfen deklariert. Im Einzelnen sind dies die Schnittstellen `LoaderHandler`, `RemoteCall`, `Skeleton`, die Klassen `LogStream` und `Operation` sowie die Ausnahmen `SkeletonMismatchException` und `SkeletonNotFoundException`.

Enthaltene Klassen

```
public class LogStream extends PrintStream
public final class ObjID extends Object implements Serializable
public class Operation extends Object
public abstract class RemoteObject extends Object implements Remote, Serializable
public abstract class RemoteServer extends RemoteObject
public abstract class RemoteStub extends RemoteObject
public class RMIClassLoader extends Object
public abstract class RMISocketFactory extends Object implements
RMIClientSocketFactory, RMIServerSocketFactory
public final class UID extends Object implements Serializable
public class UnicastRemoteObject extends RemoteServer
```

Enthaltene Schnittstellen

```
public abstract interface LoaderHandler
public abstract interface RemoteCall
public abstract interface RemoteRef extends Externalizable
public abstract interface RMIClientSocketFactory
public abstract interface RMIFailureHandler
public abstract interface RMIServerSocketFactory
public abstract interface ServerRef extends RemoteRef
public abstract interface Skeleton
public abstract interface Unreferenced
```

Enthaltene Ausnahmen

```
public class ExportException extends RemoteException
public class ServerCloneException extends CloneNotSupportedException
public class ServerNotActiveException extends Exception
public class SkeletonMismatchException extends RemoteException
public class SkeletonNotFoundException extends RemoteException
public class SocketSecurityException extends ExportException
```

java.security

Das Rückgrat des Java-Sicherheitskonzeptes.

Beschreibung

Das Paket enthält die wichtigsten Klassen, Schnittstellen und Ausnahmen des Java Securityframeworkes.

Anwendung

Die aktuelle Sicherheitsschnittstelle von Java verbindet Low-Level- und High-Level-Sicherheitsfunktionalität. So können Applets und Daten mit einer digitalen Unterschrift versehen werden, es gibt abstrakte Schnittstellen für die Verwaltung von Schlüsseln, das Zertifikatsmanagement und die Zugriffskontrolle. Spezifische API zur Unterstützung von X.509-v3-Zertifikaten und anderer Zertifikatformate sowie eine umfangreiche Funktionalität im Bereich der Zugriffskontrolle sind weitere Sicherheits-Highlights von der aktuellen Java-Plattform. Das in diesem Paket realisierte Sicherheitsframework von Java beinhaltet zahlreiche Klassen, die eine leichte Konfigurierung und eine fein abzustimmende Zugriffskontrollarchitektur implementieren. Daneben findet sich in diesem Package Support für die Generierung und Verwaltung von kryptografischen öffentlichen Schlüsselpaaren sowie eine Anzahl von exportierbaren kryptografischen Operationen inklusive Signaturgenerierung. Ein weiteres wichtiges Feature ist die Bereitstellung von signierten/geschützten Objekten und sicheren Zufallsgeneratoren. Von großer Bedeutung bei der Arbeit mit dem Sicherheitskonzept in Java ist jedoch, dass es unter der Java-2-Plattform und dem zugehörigen JDK 1.2 erheblich gegenüber den Vorgängermodellen verändert und erweitert wurde.

Die Schnittstelle `Guard` (JDK 1.2) stellt beispielsweise Methoden zum Schutz des Zugriffs auf ein Objekt zur Verfügung, `Key` (JDK 1.2) Methoden zum Definieren von der gemeinsamen Funktionalität bei allen Schlüsselobjekten, `PrivateKey` (JDK 1.2) Methoden zum Gruppieren und Bereitstellen von `PrivateKey`-Schnittstellen oder `PublicKey` (JDK 1.2) Methoden zum Gruppieren und Bereitstellen von `PublicKey`-Schnittstellen.

Unter den zahlreichen Klassen bieten beispielweise `AccessControlContext` und `AccessController` (beide JDK 1.2) Bedingungen für Zugangskontrollentscheidungen an, `AlgorithmParameterGenerator`, `AlgorithmParameterGeneratorSpi`, `AlgorithmParameters` und `AlgorithmParametersSpi` Unterstützung bei der Erstellung von Parametern oder `BasicPermission` Eigenschaften zur Bereitstellung von Basisrechten für die Zugriffskontrolle (ebenfalls neu im JDK 1.2). Ganz wichtig ist auch die Klasse `Signature`, denn diese enthält den Algorithmus für digitale Signaturen.

Die Klassen `KeyPair`, `KeyPairGenerator` und `KeyPairGeneratorSpi` beinhalten Unterstützung von Schlüsselpaaren.

Auch die zahlreichen Sicherheitsausnahmen hier in dem Paket zählen zu den Fundamenten des Java-Sicherheitsframeworks.

JDK-Version

Seit 1.1

Warnungen

Das Sicherheitsmodell von Java ist unter der Java-2-Plattform erheblich gegenüber den Vorgängerversionen verändert worden. Dies beinhaltet zum einen Erweiterungen, zum anderen aber auch massive Veränderungen des bisherigen Konzeptes, womit das Sicherheitsmodell der Java-2-Plattform in vielen Bereichen inkompatibel zu den Vorgängerkonzepten ist. Dies äußerst sich an verschiedenen Stellen.

Ein Detail ist etwa, daß der normale Java-Interpreter der Java-2-Plattform Probleme bekommen kann, wenn er Code abarbeiten soll, der sich auf das ehemalige Sicherheitmodell beruft. Um solchen Code dennoch in der Java-2-Plattform lauffähig zu halten, wurde der zusätzliche Interpreter `oldjava` bereitgestellt.

Seit der JDK-1.1-Version war im JDK-Paket ein Tool enthalten, mit welchem Java-Archive signiert werden konnten. Der Appletviewer erlaubt jedem aus dem Netz in Form einer JAR-Datei, welche als vertrauenswürdig eingestuft und mit diesem Tool entsprechend signiert wurde, geladenen Applet mit denselben Rechten auf dem lokalen Rechner zu laufen wie eine lokale Applikation. Dies hat weitreichende Konsequenzen, denn ein solches Applet ist nicht mehr Bestandteil des »Laufstalls«, in den das Java- Sicherheitsmodell Applets normalerweise zwingt. Das zugehörige Java-Sicherheitsprogramm der JDK-Version 1.1 heisst `javakey`. Das Java Security Tool ist zuständig für die Verwaltung von Datenbankentitäten, inklusive ihrer Schlüssel, Zertifikate und den Vertrauensebenen von ihnen. Dieses Programm hat sich allerdings als teilweise nicht voll befriedigend herausgestellt und ist als eine der bedeutendsten Neuerungen in der JDK-Version 1.2 durch die Programme `keytool` und `jarsigner` ersetzt worden. Daneben wurden die Sicherheitsprogramme `jar` und `policytool` hinzugefügt. Prinzipiell handelt es sich bei den besagten Tools um Programme, deren primäre Anwendung das Generieren und Verwalten von digitalen Signaturen für Archive ist. Eine Signatur verifiziert, dass eine Datei von einer bekannten Quelle kommt. Um im Java-Sicherheitskonzept eine Signatur für eine Datei zu generieren, muss zuerst ein öffentlicher Schlüssel und dann ein privater Schlüssel generiert werden, mit dem der öffentliche Schlüssel dann wieder dekodiert werden kann. `javakey` erstellt solche Schlüssel und eine Datenbank zur Verwaltung der Schlüssel und der zugeordneten Zertifikate, welche den Status der Vertrauenswürdigkeit dokumentieren. Die Anwendung der neuen Programme ist weitgehend analog. Die beiden Programme `keytool` und `jarsigner` ersetzen `javakey`

zwar vollständig und bieten eine Fülle neuer Features (etwa Schutz der Datenbank und der privaten Schlüssel mit Passwörtern). Die beiden Programme sind jedoch ausdrücklich nicht abwärtskompatibel zu dem Keystore- und Datenbankformat, welches von `javakey` im JDK 1.1 verwendet wurde.

Ein Resultat der zahlreichen Veränderungen im Sicherheitmodell ist, dass eine große Menge von Elementen aus dem bisherigen Modell unter der Java-2-Plattform als deprecated gelten. Im Einzelnen sind das viele einzelne Methoden und Variablen innerhalb verschiedenster Klassen und Schnittstellen, aber auch die ganze Schnittstelle `Certificate` sowie die Klassen `Identity`, `IdentityScope` und `Signer`.

Enthaltene Klassen

```
public final class AccessControlContext extends Object
public final class AccessController extends Object
public class AlgorithmParameterGenerator extends Object
public abstract class AlgorithmParameterGeneratorSpi extends Object
public class AlgorithmParameters extends Object
public abstract class AlgorithmParametersSpi extends Object
public final class AllPermission extends Permission
public abstract class BasicPermission extends Permission implements Serializable
public class CodeSource extends Object implements Serializable
public class DigestInputStream extends FilterInputStream
public class DigestOutputStream extends FilterOutputStream
public class GuardedObject extends Object implements Serializable
public abstract class Identity extends Object implements Principal, Serializable
public abstract class IdentityScope extends Identity
public class KeyFactory extends Object
public abstract class KeyFactorySpi extends Object
public final class KeyPair extends Object implements Serializable
public abstract class KeyPairGenerator extends KeyPairGeneratorSpi
public abstract class KeyPairGeneratorSpi extends Object
public class KeyStore extends Object
public abstract class KeyStoreSpi extends Object
public abstract class MessageDigest extends MessageDigestSpi
public abstract class MessageDigestSpi extends Object
public abstract class Permission extends Object implements Guard, Serializable
public abstract class PermissionCollection extends Object implements Serializable
public final class Permissions extends PermissionCollection implements
Serializable
public abstract class Policy extends Object
public class ProtectionDomain extends Object
public abstract class Provider extends Properties
public class SecureClassLoader extends ClassLoader
public class SecureRandom extends Random
public abstract class SecureRandomSpi extends Object implements Serializable
public final class Security extends Object
public final class SecurityPermission extends BasicPermission
public abstract class Signature extends SignatureSpi
public abstract class SignatureSpi extends Object
```

```
public final class SignedObject extends Object implements Serializable
public abstract class Signer extends Identity
public final class UnresolvedPermission extends Permission implements Serializable
```

Enthaltene Schnittstellen

```
public abstract interface Certificate
public abstract interface Guard
public abstract interface Key extends Serializable
public abstract interface Principal
public abstract interface PrivateKey extends Key
public abstract interface PrivilegedAction
public abstract interface PrivilegedExceptionAction
public abstract interface PublicKey extends Key
```

Enthaltene Ausnahmen

```
public class AccessControlException extends SecurityException
public class DigestException extends GeneralSecurityException
public class GeneralSecurityException extends Exception
public class InvalidAlgorithmParameterException extends GeneralSecurityException
public class InvalidKeyException extends KeyException
public class InvalidParameterException extends IllegalArgumentException
public class KeyException extends GeneralSecurityException
public class KeyManagementException extends KeyException
public class KeyStoreException extends GeneralSecurityException
public class NoSuchAlgorithmException extends GeneralSecurityException
public class NoSuchProviderException extends GeneralSecurityException
public class PrivilegedActionException extends Exception
public class ProviderException extends RuntimeException
public class SignatureException extends GeneralSecurityException
public class UnrecoverableKeyException extends GeneralSecurityException
```

java.security.acl

Ergänzung des Sicherheitskonzeptes von Java.

Beschreibung

Das Paket enthält ergänzende Wächterfunktionalität für das Sicherheitskonzept von Java.

Anwendung

Das Paket enhält nur Schnittstellen und Ausnahmen (keine Klassen), mit denen Zugriffe auf Ressourcen durch Wächterfunktionen kontrolliert werden können. `Acl` ist ein Interface, welches eine Zugriffskontroll-Liste (Access Control List = ACL) darstellt, `AclEntry` verfügt über Methoden zum Hinzufügen, Entfernen oder Setzen von Eigenschaften für die Objekte von jeder ACLEntry in dem ACL. `Group` enthält Methoden zum Hinzufügen oder Entfernen eines Mitglieds zu einer Gruppe von Objekten, `Owner` repräsentiert den Eigentümer von einem ACL und `Permission` den Typ von gewährtem Zugriff.

JDK-Version

Seit 1.1

Enthaltene Klassen

(keine)

Enthaltene Schnittstellen

```
public abstract interface Acl extends Owner
public abstract interface AclEntry extends Cloneable
public abstract interface Group extends Principal
public abstract interface Owner
public abstract interface Permission
```

Enthaltene Ausnahmen

```
public class AclNotFoundException extends Exception
public class LastOwnerException extends Exception
public class NotOwnerException extends Exception
```

java.security.cert

Ergänzung des Sicherheitskonzeptes von Java.

Beschreibung

Das Paket beinhaltet Klassen und Interfaces zum Erstellen und Verwalten von Zertifikaten.

Anwendung

Wesentliche Anwendung von diesem Paket ist die Unterstützung von X.509-v3-Zertifikaten. Das einzige Interface des Paketes – `X509Extension` – sagt beispielsweise schon über seinen Namen aus, dass es dazu die wichtigsten Methoden enthält. Dazu gibt es mit den diese Schnittstelle implementierenden abstrakten Klassen `X509Certificate`, `X509CRL` und `X509CRLEntry` direkte Anwendungen davon.

JDK-Version

Seit 1.2

Enthaltene Klassen

```
public abstract class Certificate extends Object
public class CertificateFactory extends Object
public abstract class CertificateFactorySpi extends Object
public abstract class CRL extends Object
public abstract class X509Certificate extends Certificate implements X509Extension
public abstract class X509CRL extends CRL implements X509Extension
public abstract class X509CRLEntry extends Object implements X509Extension
```

Enthaltene Schnittstellen

```
public abstract interface X509Extension
```

Enthaltene Ausnahmen

```
public class CertificateEncodingException extends CertificateException
public class CertificateException extends GeneralSecurityException
public class CertificateExpiredException extends CertificateException
public class CertificateNotYetValidException extends CertificateException
public class CertificateParsingException extends CertificateException
public class CRLException extends GeneralSecurityException
```

java.security.interfaces

Ergänzung des Sicherheitskonzeptes von Java.

Beschreibung

Unterstützung für die RSA- und DSA-Technologie.

Anwendung

Im Rahmen des Java-Sicherheitskonzeptes kann die RSA- und DSA-Technologie genutzt werden. Dieses Paket enthält Schnittstellen, die dies gewährleisten. Etwa `DSAKey` mit Methoden für die Verwendung in beglaubigten Komponenten inklusive Java-Applets und ActiveX-Controls, `DSAKeyPairGenerator` mit Methoden zum Generieren von einem DSA-Schlüsselpaars `DSAParams` mit Methoden zum Zugriff auf DSA-Parameter, `DSAPrivateKey` für private DSA-Schlüssel oder `DSAPublicKey` für öffentliche DSA-Schlüssel.

JDK-Version

Seit 1.1

Enthaltene Klassen

(keine)

Enthaltene Schnittstellen

```
public abstract interface DSAKey
public abstract interface DSAKeyPairGenerator
public abstract interface DSAParams
public abstract interface DSAPrivateKey extends DSAKey, PrivateKey
public abstract interface DSAPublicKey extends DSAKey, PublicKey
public abstract interface RSAPrivateCrtKey extends RSAPrivateKey
public abstract interface RSAPrivateKey extends PrivateKey
public abstract interface RSAPublicKey extends PublicKey
```

Enthaltene Ausnahmen

(keine)

java.security.spec

Ergänzung des Sicherheitskonzeptes von Java.

Beschreibung

Klassen und Schnittstellen für die Spezifikation von Schlüsseln und Algorithmen-Parametern.

Anwendung

Eine Schlüsselspezifikation ist eine transparente Repräsentation des Materials, das den Schlüssel bildet. Ein Schlüssel kann über einen Algorithmus oder aber auch einen Algorithmus-unabhängigen Weg erstellt werden. Das Package beinhaltet Schlüsselspezifikationen für DSA-Formate (öffentliche und private Schlüssel), RSA-Formate (öffentliche und private Schlüssel), private Schlüssel nach PKCS-#8-Norm im DER-verschlüsselten Format und öffentliche und private Schlüssel nach X.509-Norm im DER-verschlüsselten Format. Konkret beinhaltet etwa die Schnittstelle `DSAParameterSpec` Parameter für die Verwendung mit dem DSA-Algorithmus, `DSAPrivateKeySpec` private DSA-Schlüssel und zugehörige Parameter, `DSAPublicKeySpec` öffentliche DSA-Schlüssel und zugehörige Parameter und `PKCS8EncodedKeySpec` DER-Verschlüsselung von privaten Schlüssel im PKCS-#8-Standard sowie `X509EncodedKeySpec` DER-Verschlüsselung von privaten oder öffentlichen Schlüsseln im X.509-Standard.

Eine Algorithmus-Parameterspezifikation ist eine transparente Repräsentation von einem Set von Parametern, welche im Zusammenhang mit einem Algorithmus genutzt werden. Dieses Package beinhaltet eine Spezifikation für Parameter, welche im Zusammenhang mit dem DSA-Algorithmus genutzt werden. Etwa die Schnittstelle `AlgorithmParameterSpec` mit Methoden zum Spezifizieren von kryptografischen Parametern.

JDK-Version

Seit 1.2

Enthaltene Klassen

```
public class DSAParameterSpec extends Object implements AlgorithmParameterSpec,
DSAParams
public class DSAPrivateKeySpec extends Object implements KeySpec
public class DSAPublicKeySpec extends Object implements KeySpec
public abstract class EncodedKeySpec extends Object implements KeySpec
public class PKCS8EncodedKeySpec extends EncodedKeySpec
public class RSAPrivateCrtKeySpec extends RSAPrivateKeySpec
public class RSAPrivateKeySpec extends Object implements KeySpec
public class RSAPublicKeySpec extends Object implements KeySpec
public class X509EncodedKeySpec extends EncodedKeySpec
```

Enthaltene Schnittstellen

```
public abstract interface AlgorithmParameterSpec
public abstract interface KeySpec
```

Enthaltene Ausnahmen

```
public class InvalidKeySpecException extends GeneralSecurityException
public class InvalidParameterSpecException extends GeneralSecurityException
```

java.sql

Der Hauptpfeiler der Datenbankfunktionalität von Java.

Beschreibung

Die wesentliche Unterstützung für den Datenbankzugriff unter Java mittels JDBC.

Anwendung

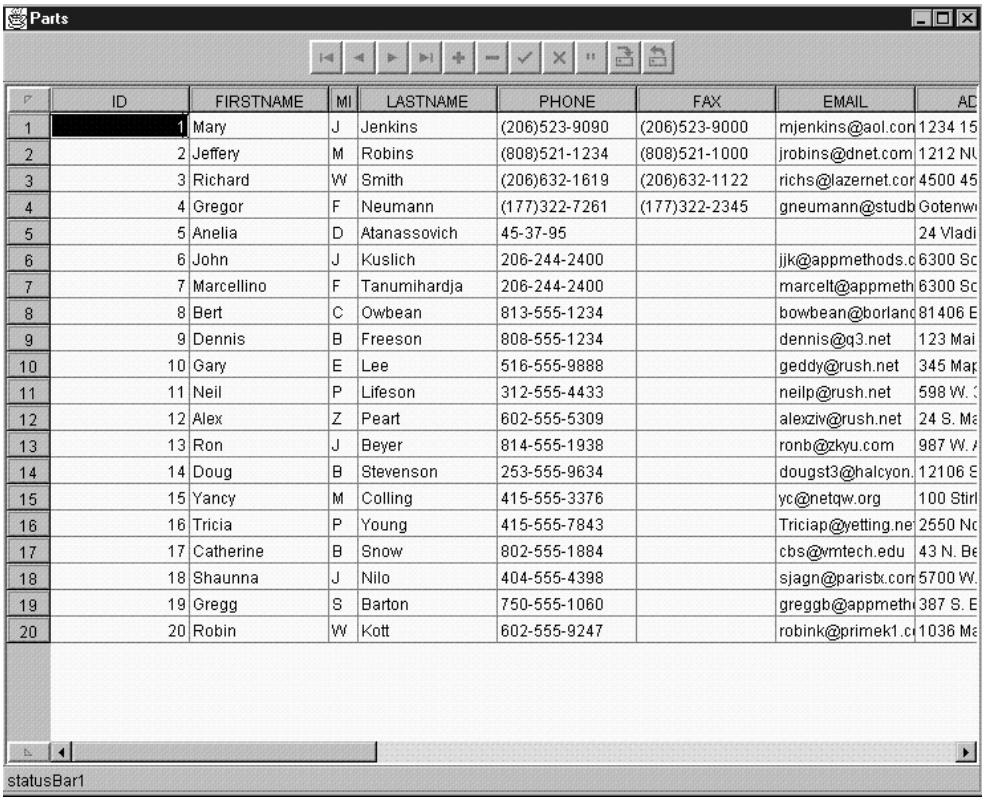
Die netzwerkorientierte Struktur von Java macht die Sprache zu einem idealen Kandidaten für Client-Server-Einsätze, etwa Client-Server-Datenbanken. Für den prinzipiellen Zugang zu Datenbanken von außen (ob lokal oder ein Netzwerk) bietet Java eine einfache Möglichkeit – die Java DataBase Connectivity (JDBC). Diese Schnittstelle (unter der Java-2-Plattform JDBC 2.0) erlaubt eine datenbankunabhängige Entwicklung von Java-Clients, die auf zahlreiche verbreitete relationale Datenbanken zugreifen können.

Aufgrund des modularen Aufbaus der JDBC-Spezifikation können aufgabenbezogene Erweiterungen hinzugefügt und für die Datenverarbeitung notwendige Tools jederzeit integriert werden. Diese können über den JDBC-Layer mit den erforderlichen Datenbanken kommunizieren. Die JDBC-Treiber übernehmen dabei die gesamte Datenbankanbindung. Sofern sich Datenbankentwickler an die normale Standard-SQL-Syntax (SQL = Structured Query Language – eine universelle Datenbanksprache für Aktionen auf relationalen Datenbanken) halten, sollte jedes Datenbankprodukt mit einem JDBC-kompatiblen Treiber verwendet werden können.

JDBC ist nicht als Produkt zu verstehen, sondern es handelt sich um die abstrakte Spezifikation einer Schnittstelle zwischen einer Client-Anwendung und einer SQL-Schnittstelle. Das Interface ist als Low-Level-API zum grundlegenden SQL-Zugriff entworfen worden. Es liegt an den verschiedenen Herstellern von Datenbanken und Software, ob JDBC-kompatible Treiber eingebaut werden, damit Java-Anwendungen mit den Datenbanksystemen verbunden werden können. Einer der ersten JDBC-kompatiblen Treiber war Microsofts JDBC-Treiber für ODBC-kompatible Datenbanken.

JDBC arbeitet auf zwei Stufen. Die erste Stufe ist der Verbindungsaufbau zwischen Java-Anwendung und JDBC-Treibermanager mittels der JDBC-API. Über diesen JDBC-Treibermanager kann ein Java-Programm dann mehrere JDBC-Treiber verwalten und mit ihnen Informationen und Daten austauschen. Jeder Treiber wiederum kann beispielsweise aus Java direkt auf lokale Daten zugreifen, ODBC als Zwischenebene dazwischenschalten oder einen Netzwerkzugriff auf eine Datenbank auslösen. Die Treiber registrieren sich bei dem JDBC-Manager während der Initialisierung, so dass der Manager einen Überblick über alle verfügbaren Treiber hat. Der JDBC-Manager hat noch weitergehende Aufgaben. So untersucht er

beispielsweise den Zustand eines Treibers, damit gegebenenfalls ein Applet einen geeigneten Treiber herunterladen kann, wenn noch keiner auf dem System vorhanden ist.



	ID	FIRSTNAME	MI	LASTNAME	PHONE	FAX	EMAIL	AD
1	1	Mary	J	Jenkins	(206)523-9090	(206)523-9000	mjenkins@aol.com	1234 15
2	2	Jeffery	M	Robins	(808)521-1234	(808)521-1000	jrobins@dnet.com	1212 NU
3	3	Richard	W	Smith	(206)632-1619	(206)632-1122	richs@lazernet.cor	4500 45
4	4	Gregor	F	Neumann	(177)322-7261	(177)322-2345	gneumann@studb	Gotenw
5	5	Anelia	D	Atanassovich	45-37-95			24 Vladi
6	6	John	J	Kuslich	206-244-2400		jjk@appmethods.c	6300 Sc
7	7	Marcellino	F	Tanumihardja	206-244-2400		marcelt@appmeth	6300 Sc
8	8	Bert	C	Owbean	813-555-1234		bowbean@borlanc	81406 E
9	9	Dennis	B	Freeson	808-555-1234		dennis@q3.net	123 Mai
10	10	Gary	E	Lee	516-555-9888		geddy@rush.net	345 Map
11	11	Neil	P	Lifeson	312-555-4433		neilp@rush.net	598 W.
12	12	Alex	Z	Peart	602-555-5309		alexziv@rush.net	24 S. Ma
13	13	Ron	J	Beyer	814-555-1938		ronb@zkyu.com	987 W. A
14	14	Doug	B	Stevenson	253-555-9634		dougst3@halcyon.	12106 S
15	15	Yancy	M	Colling	415-555-3376		yc@netqw.org	100 Stirl
16	16	Tricia	P	Young	415-555-7843		Triciap@yetting.ne	2550 Nc
17	17	Catherine	B	Snow	802-555-1884		cbs@vmtech.edu	43 N. Be
18	18	Shaunna	J	Nilo	404-555-4398		sjagn@paristx.com	5700 W.
19	19	Gregg	S	Barton	750-555-1060		greggb@appmeth	387 S. E
20	20	Robin	W	Kott	602-555-9247		robink@primek1.c	1036 Ma

Eine Java-Datenbankapplikation mit JDBC und SQL-Zugriff.

Während des Versuchs, sich mit einer Datenbank zu verbinden, gibt das Java-Programm einen Datenbank-URL an den JDBC-Manager weiter. Der JDBC-Manager ruft dann jeden geladenen JDBC-Treiber, bis man den angefragten URL öffnen kann. Jeder Treiber ignoriert solche URLs, die Datenbanken erfordern, zu denen er sich nicht verbinden kann. Java Client-Programme können diesen Vorgang des Treibersuchens übergehen und explizit angeben, welcher Treiber verwendet werden soll, wenn der Java-Client bereits vorher weiß, welcher Treiber geeignet ist. Die URLs von JDBC haben immer die folgende Form: `jdbc:subprotocol:subname`.

Das Subprotokoll ist der Name des Verbindungsprotokolls, und der Subname ist der Name der jeweiligen Datenbank innerhalb der Domäne des Protokolls. Sofern über den Subnamen Informationen über Host und Port verschlüsselt sind, sollte dieser den Host und den Port in der URL-Standardnotation angeben:

```
//hostname:port/subname
```

JDBC besteht aus mehreren portablen Java-Klassen/Schnittstellen, die allesamt zum Paket `java.sql` gehören. Etwa `java.sql.DriverManager`, welche den JDBC-Treibermanager repräsentiert, `java.sql.Driver` - die Schnittstelle eines abstrakten JDBC-Treibers, `java.sql.Connection` – eine verbindungsspezifische Schnittstelle, `java.sql.Statement` – eine Container-Schnittstelle für SQL-Anweisungen, oder `java.sql.ResultSet` – eine Zugriffsverwaltung der Ergebnisse. Daneben werden in dem Paket zahlreiche Meta-Informationen verwaltet – etwa über `java.sql.DatabaseMetaData` Meta-Informationen über die Datenbank oder über `java.sql.ResultSetMetaData` Meta-Informationen über die Ergebnisse.

JDK-Version

Seit 1.1

Enthaltene Klassen

```
public class Date extends Date
public class DriverManager extends Object
public class DriverPropertyInfo extends Object
public class Time extends Date
public class Timestamp extends Date
public class Types extends Object
```

Enthaltene Schnittstellen

```
public abstract interface Array
public abstract interface Blob
public abstract interface CallableStatement extends PreparedStatement
public abstract interface Clob
public abstract interface Connection
public abstract interface DatabaseMetaData
public abstract interface Driver
public abstract interface PreparedStatement extends Statement
public abstract interface Ref
public abstract interface ResultSet
public abstract interface ResultSetMetaData
public abstract interface SQLData
public abstract interface SQLInput
public abstract interface SQLOutput
public abstract interface Statement
public abstract interface Struct
```

Enthaltene Ausnahmen

```
public class BatchUpdateException extends SQLException
public class DataTruncation extends SQLWarning
public class SQLException extends Exception
public class SQLWarning extends SQLException
```

Verwandte Pakete

`java.net.URL`
`java.util.*`

java.text

Erweiterte Textunterstützung in Java.

Beschreibung

Das Paket stellt Klassen und Interfaces zum Handling von Text, Datum, Zahlen und Nachrichten in einer von natürlichen Sprachen unabhängigen Weise zur Verfügung. Das bedeutet, eine Applikation oder ein Applet kann sprachunabhängig erstellt werden und bei Bedarf dynamisch Ressourcen mit lokalen Informationen hinzulinken.

Anwendung

Die Klassen in diesem Paket beinhalten Möglichkeiten zum Formatieren von einem Datum, von Zahlen und Nachrichten, zum Suchen in Strings, zum Sortieren in Strings, für Parse-Prozesse usw. So beinhalten die Schnittstellen `AttributeCharacterIterator` Methoden zum Durchforsten von Text und `CharacterIterator` und Methoden zum Analysieren eines Strings und der Rückgabe diverser Informationen über ihn.

Die Klasse `BreakIterator` stellt Eigenschaften zur Lokalisierung von Textbegrenzern bereit und `Collation` dient zum Vergleichen von Unicodetext.

Mit der Klasse `Format` gibt es in dem Paket eine Basisklasse für alle Formate, die zahlreiche Spezialisierungen in dem Paket besitzt. `DateFormat` ist beispielsweise eine abstrakte Klasse mit einigen date-time-Formatierungs-Subklassen und `DateFormatSymbols` beinhaltet zahlreiche lokale date-time-Formatierungsdaten.

Über `DecimalFormat` und `DecimalFormatSymbols` stehen Methoden zum Formatieren von Zahlen beziehungsweise Symbole für die Verwendung des Dezimalformats bei der Formatierung von Zahlen zur Verfügung. `MessageFormat` dagegen stellt Methoden zum Erstellen von formatierten Nachrichten bereit und `NumberFormat` dient als abstrakte Klasse für alle Zahlenformate.

JDK-Version

Seit 1.1

Warnungen

Dieses Paket hatte einige neue Elemente und Strukturen in den Betaversionen des JDK 1.2 eingeführt, welche für die Finalversion wieder verworfen wurden. Erst recht gegenüber der Version 1.1 sind massive Veränderungen zu beachten.

Enthaltene Klassen

```
public class Annotation extends Object
public static class AttributedCharacterIterator.Attribute extends Object
implements Serializable
public class AttributedString extends Object
public abstract class BreakIterator extends Object implements Cloneable
public class ChoiceFormat extends NumberFormat
public final class CollationElementIterator extends Object
public final class CollationKey extends Object implements Comparable
public abstract class Collator extends Object implements Comparator, Cloneable
public abstract class DateFormat extends Format
public class DateFormatSymbols extends Object implements Serializable, Cloneable
public class DecimalFormat extends NumberFormat
public final class DecimalFormatSymbols extends Object implements Cloneable,
Serializable
public class FieldPosition extends Object
public abstract class Format extends Object implements Serializable, Cloneable
public class MessageFormat extends Format
public abstract class NumberFormat extends Format
public class ParsePosition extends Object
public class RuleBasedCollator extends Collator
public class SimpleDateFormat extends DateFormat
public final class StringCharacterIterator extends Object implements
CharacterIterator
```

Enthaltene Schnittstellen

```
public abstract interface AttributedCharacterIterator extends CharacterIterator
public abstract interface CharacterIterator extends Cloneable
```

Enthaltene Ausnahmen

```
public class ParseException extends Exception
```

java.util

Zahlreiche Klassen und Schnittstellen für die unterschiedlichsten ergänzenden Funktionalitäten.

Beschreibung

Hierbei handelt es sich um ein Paket mit verschiedenen Utility-Klassen und -Schnittstellen, etwa Zufallszahlen oder Systemeigenschaften.

Anwendung

Zahlreiche Funktionalitäten sind nicht umfangreich genug, um sie in einem spezifischen Paket zusammenzufassen. Diese lassen sich aber gut einem allgemeinen Paket mit Hilfstechnologien zuordnen. Das Paket `java.util` beinhaltet beispielsweise die extrem wichtige Schnittstelle `EventListener` mit Methoden zum Beobachten von Ereignissen oder `Map` mit Methoden zum Mappen von Schlüsseln in Werte.

Die Klassen gehen von `Arrays` mit Eigenschaften für eine Array-Sortierung und anderer Manipulationen über `Calendar` (ein allgemeiner Kalender), `Date` (das aktuelle Systemdatum sowie Methoden zum Generieren und Abgleichen von Datumsangaben) und `GregorianCalendar` (eine Repräsentation des Gregorianischen Kalenders) über `HashMap` (die Implementation von einer Map – basierend auf Hash-Tabellen) und `HashSet` (eine Implementation von einem Set – basierend auf Hash-Tabellen), sowie `Hashtable` (eine Hash-Tabelle) bis `Random` (Utilitys für die Erzeugung von Zufallszahlen) und `Vector` (ein Array von Objekten, welches dynamisch wachsen kann).

JDK-Version

Seit 1.0

Warnungen

Insbesondere viele Methoden der Klasse `Date` gelten mittelweile als deprecated. Etwa vier der bisherigen `Date()`-Konstruktoren sowie zahlreiche Methoden: `UTC()`, `parse()`, `getYear()`, `setYear()`, `getMonth()`, `setMonth()`, `getDate()`, `setDate()`, `getDay()`, `getHours()`, `setHours()`, `getMinutes()`, `setMinutes()`, `getSeconds()`, `setSeconds()`, `toLocaleString()`, `toGMTString()`, und `getTimeZoneOffset()`.

Enthaltene Klassen

```
public abstract class AbstractCollection extends Object implements Collection
public abstract class AbstractList extends AbstractCollection implements List
public abstract class AbstractMap extends Object implements Map
public abstract class AbstractSequentialList extends AbstractList
public abstract class AbstractSet extends AbstractCollection implements Set
```

```
public class ArrayList extends AbstractList implements List, Cloneable,
Serializable
public class Arrays extends Object
public class BitSet extends Object implements Cloneable, Serializable
public abstract class Calendar extends Object implements Serializable, Cloneable
public class Collections extends Object
public class Date extends Object implements Serializable, Cloneable, Comparable
public abstract class Dictionary extends Object
public class EventObject extends Object implements Serializable
public class GregorianCalendar extends Calendar
public class HashMap extends AbstractMap implements Map, Cloneable, Serializable
public class HashSet extends AbstractSet implements Set, Cloneable, Serializable
public class Hashtable extends Dictionary implements Map, Cloneable, Serializable
public class LinkedList extends AbstractSequentialList implements List, Cloneable,
Serializable
public abstract class ListResourceBundle extends ResourceBundle
public final class Locale extends Object implements Cloneable, Serializable
public class Observable extends Object
public class Properties extends Hashtable
public final class PropertyPermission extends BasicPermission
public class PropertyResourceBundle extends ResourceBundle
public class Random extends Object implements Serializable
public abstract class ResourceBundle extends Object
public class SimpleTimeZone extends TimeZone
public class Stack extends Vector
public class StringTokenizer extends Object implements Enumeration
public abstract class TimeZone extends Object implements Serializable, Cloneable
public class TreeMap extends AbstractMap implements SortedMap, Cloneable,
Serializable
public class TreeSet extends AbstractSet implements SortedSet, Cloneable,
Serializable
public class Vector extends AbstractList implements List, Cloneable, Serializable
public class WeakHashMap extends AbstractMap implements Map
```

Enthaltene Schnittstellen

```
public abstract interface Collection
public abstract interface Comparator
public abstract interface Enumeration
public abstract interface EventListener
public abstract interface Iterator
public abstract interface List extends Collection
public abstract interface ListIterator extends Iterator
public abstract interface Map
public abstract static interface Map.Entry
public abstract interface Observer
public abstract interface Set extends Collection
public abstract interface SortedMap extends Map
public abstract interface SortedSet extends Set
```

Enthaltene Ausnahmen

```
public class ConcurrentModificationException extends RuntimeException
public class EmptyStackException extends RuntimeException
public class MissingResourceException extends RuntimeException
public class NoSuchElementException extends RuntimeException
public class TooManyListenersException extends Exception
```

java.util.jar

Unterstützung des jar-Formates (ein spezielles Java-Komprimierungsformat).

Beschreibung

Hierbei handelt es sich um ein Paket für die Behandlung von Java Archive Resource (JAR)-Dateien.

Anwendung

Die Behandlung von JAR-Dateien (das bedeutet, in dem spezifischen Java-Format jar komprimierte Dateien) erfolgt über eine Ausnahme und diverse Klassen, die unter anderem das Mapping von manifestierten Attributnamen in zugehörige String-Werte (`Attribute`) erlauben, einen Attributnamen in dieser Map repräsentieren oder Eigenschaften zum Lesen von JAR-Dateien (`JarFile`) oder Eingabeströme (`JarInputStream`) beziehungsweise Ausgabeströme (`JarOutputStream`) für JAR-Dateien zur Verfügung stellen.

JDK-Version

Seit 1.2

Enthaltene Klassen

```
public class Attributes extends Object implements Map, Cloneable
public static class Attributes.Name extends Object
public class JarEntry extends ZipEntry
public class JarFile extends ZipFile
public class JarInputStream extends ZipInputStream
public class JarOutputStream extends ZipOutputStream
public class Manifest extends Object implements Cloneable
```

Enthaltene Schnittstellen

(keine)

Enthaltene Ausnahmen

```
public class JarException extends ZipException
```

java.util.zip

Unterstützung von Komprimierungsvorgängen.

Beschreibung

Dieses Paket enthält Klassen für den Zugriff auf komprimierte Dateiarchive (Zip- und gZip-Algorithmus).

Anwendung

In diesem Paket befinden sich Klassen/Schnittstellen für Lese- und Schreibzugriffe auf komprimierte Dateiarchive im Standard-Zip- und -gZip-Format. Ebenfalls enthalten sind Klassen zum Komprimieren und Dekomprimieren von Daten mit dem DEFLATER-Kompressionsalgorithmus und einige Utility-Klassen für die Verwendung von CRC-32- und Adler-32-Checksummen bei Eingabeströmen. Etwa die Schnittstelle `Checksum` mit allgemeinen Methoden für die Berechnung einer Checksumme oder die Klassen `Adler32` zur Berechnung einer Adler-32-Checksumme und `CRC32` zur Berechnung einer CRC-32-Checksumme.

JDK-Version

Seit 1.1

Enthaltene Klassen

```
public class Adler32 extends Object implements Checksum
public class CheckedInputStream extends FilterInputStream
public class CheckedOutputStream extends FilterOutputStream
public class CRC32 extends Object implements Checksum
public class Deflater extends Object
public class DeflaterOutputStream extends FilterOutputStream
public class GZIPInputStream extends InflaterInputStream
public class GZIPOutputStream extends DeflaterOutputStream
public class Inflater extends Object
public class InflaterInputStream extends FilterInputStream
public class ZipEntry extends Object implements java.util.zip.ZipConstants,
Cloneable
public class ZipFile extends Object implements java.util.zip.ZipConstants
public class ZipInputStream extends InflaterInputStream implements
java.util.zip.ZipConstants
public class ZipOutputStream extends DeflaterOutputStream implements
java.util.zip.ZipConstants
```

Enthaltene Schnittstellen

```
public abstract interface Checksum
```

Enthaltene Ausnahmen

```
public class DataFormatException extends Exception
public class ZipException extends IOException
```

javax.accessibility

Allgemeine Unterstützung für den Zugriff auf verschiedene Benutzerschnittstellen.

Beschreibung

Das Paket definiert die Bedingungen zwischen Benutzerschnittstellenkomponenten und ergänzenden Hilfstechnologien.

Anwendung

Mit dem in diesem Paket im Wesentlichen realisierten Java-Accessibility-Konzept können Java-Applikationen generiert werden, die mit innovativen Hilfstechnologien (etwa Spracheingabesystemen oder Blindensprache-Terminals) interagieren können. So stellt die Schnittstelle `Accessible` Methoden bereit, um ein Benutzerschnittstellen-Element für die verschiedenen Zugriffsmöglichkeiten zugänglich zu machen oder `AccessibleText` Methoden, um Text exakt an einer angegebenen Koordinate auf einem grafischen Ausgabegerät auszugeben. Die Klasse `AccessibleRole` enthält eine präzise Beschreibung von Regeln, die ein Element in einem beliebigen Userinterface innerhalb der Anwenderschnittstelle einhalten muss, `AccessibleState` eine präzise Beschreibung des Status von einem Element in einem Userinterface und `AccessibleStateSet` den Satz von allen `AccessibleState`-Objekten, welche den vollständigen Status von einem Schnittstellenelement repräsentieren.

Die daraus resultierenden Applikationen sind nicht auf bestimmte technische Plattformen beschränkt, sondern können auf jeder Maschine eingesetzt werden, welche die virtuelle Javamaschine unterstützt. Das neue Java-Accessibility-API ist eines der Kernbestandteile der Java Foundation Classes.

JDK-Version

Seit 1.2

Warnungen

Das Accessibility-Konzept gab es bereits vor der Finalversion 1.2 des JDK, aber die Finalversion des JDK 1.2 hat bezüglich des Accessibility-Packages eine erhebliche Veränderung gegenüber den Vorgängerversionen (auch den Betaversionen des JDK 1.2 bis Beta 3) gebracht. Das vorher dem Namensraum `com.sun.java.*` (oder kurz `java.*`) zugeordnete Paket wurde in dem Namensraum `javax.*` verlagert. Außerdem wurde die Paketstruktur erheblich verändert.

Enthaltene Klassen

```
public abstract class AccessibleBundle extends Object
public abstract class AccessibleContext extends Object
public abstract class AccessibleHyperlink extends Object implements
AccessibleAction
public class AccessibleResourceBundle extends ListResourceBundle
public class AccessibleRole extends AccessibleBundle
public class AccessibleState extends AccessibleBundle
public class AccessibleStateSet extends Object
```

Enthaltene Schnittstellen

```
public abstract interface Accessible
public abstract interface AccessibleAction
public abstract interface AccessibleComponent
public abstract interface AccessibleHypertext extends AccessibleText
public abstract interface AccessibleSelection
public abstract interface AccessibleText
public abstract interface AccessibleValue
```

Enthaltene Ausnahmen

(keine)

javax.swing

Das Basispaket für das Swing-Konzept.

Beschreibung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Das Look and Feel von Java-Benutzerschnittstellen passt sich damit immer mehr den Standards der GUI-Welt an. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet.

Anwendung

Swing sorgt zum einen dafür, dass sich das Look and Feel von Java-Benutzerschnittstellen immer mehr den Standards der GUI-Welt anpasst. Das Swing-Konzept geht jedoch zum anderen noch weiter. Swing ist vollständig in reinem Java entwickelt und dort ohne Widersprüche implementiert. Swing basiert auf dem JDK 1.1 Lightweight UI Framework und versetzt Entwickler in die Lage, einen Satz von GUI-Komponenten zu entwickeln, welche sich automatisch – auf Wunsch und zur Laufzeit – dem passenden Look and Feel für jede Betriebssystemplattform (Windows, Solaris, Macintosh) anpassen. Swing-Komponenten beinhalten alle bisher existierenden AWT-Komponentensätze (Button, Scrollbar, Label, etc.) sowie einen großen Satz von »High-Level-Komponenten« (etwa Baumansicht, Listboxen, usw.). Der Bestand an Schnittstellen und Klassen deckt jede der grundlegenden Funktionalitäten des Swing-Konzeptes ab. Die Swing-Komponenten beinhalten Duplikate von allen AWT-Komponenten von Java 1.1 und unzählige zusätzliche Komponenten. Von Schnittstellen wie `Action` mit Methoden in Verbindung mit verschiedenen Komponenten im Rahmen der `action`-Kommandofunktionalität angefangen über Klassen für jede Swing-AWT-Komponente (beispielsweise `Japplet`-Swing – Applet, `JButton`-Swing – anklickbarer Button, `JCheckBox`-Swing – Checkbox, `JCheckBoxMenuItem`-Swing – Checkbox-Eintrag in einem Pull-down-Menü, `JColorChooser`-Swing – Farbauswahldialog, `JComboBox`-Swing – kombinierte Listbox und Textfeld, `JComponent`-Swing – Komponenten-Ursprungsklasse, `JDesktopPane`-Swing – Desktoppane, `JDialog`-Swing – Dialogfenster, `JEditorPane`-Swing – Editorpane, `JFileChooser`-Swing – Dateiauswahldialog, `JFrame`-Swing – Framefenster, `JinternalFrame`-Swing – internes Frame, `JLabel`-Swing – Label, `JLayeredPane`-Swing – schichtenweiser Panecontainer, `JList`-Swing – Auswahlliste, `JMenu`-Swing – Pull-down-Menü, `JMenuBar`-Swing – Menubar in einer Fensterkomponente, `JMenuItem`-Swing – Menüeintrag in einem Pull-down-Menü, `JOptionPane`-Swing – Options-Panecontainer, `JPanel`-Swing – Panelcontainer, `JPasswordField`-Swing – Passwort-

Textfeld mit verdeckten Zeichen, JPopupMenu-Swing – Pop-up-Menü, JprogressBar-Swing – Progressbar-Komponente, JRadioButton-Swing – Radiobutton, JRadioButtonMenuItem-Swing – Radiobutton in einem Pull-down-Menü, JRootPane-Swing – Rootpanecontainer, JScrollBar-Swing – Bildlaufleiste, JScrollPane-Swing – scrollbarer Panecontainer, JSeparator Swing – Menü-Separatorkomponente, JSlider-Swing – Slider, JSplitPane-Swing – geteilter Panecontainer, JTabbedPane-Swing – Panecontainer mit Tabulator, JTable-Swing – Tabelle, JTextArea-Swing – Multiline-Textfeld) bis zu der UIManager-Klasse zum Kennzeichnen des aktuellen Look and Feel.

JDK-Version

Seit 1.1

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert. Wurde das Swing-Paket in den Vorgängerversionen des JDK 1.2-Final noch unter java.awt.swing geführt, ist es jetzt – wie zahlreiche damit verbundene Paket – in der javax-Struktur eingeordnet. Insgesamt ist die Struktur des Swing-API vollkommen überarbeitet worden und hat vielfach nichts mehr gemein mit dem Konzept der Betaversionen und den 1.1x-Versionen. Zahlreiche Klassen und Schnittstellen, welche in den Beta-Ausführungen vorgestellt wurden, sind im Finalrelease nicht realisiert. Entweder ist deren Funktionalität in anderen Paketen und Klassen aufgegangen oder die Realisierung hat sich als nicht sinnvoll herausgestellt. Es wird deshalb zu vielen Inkompatibilitäten kommen, wenn Applikationen, welche mit einem Beta-JDK 1.2 erstellt wurden, mit dem neuen JDK ausgeführt oder überarbeitet werden sollen.

Enthaltene Klassen

```
public abstract class AbstractAction extends Object implements Action, Cloneable,
Serializable
public abstract class AbstractButton extends JComponent implements ItemSelectable,
SwingConstants
public abstract class AbstractListModel extends Object implements ListModel,
Serializable
public class BorderFactory extends Object
public class Box extends Container implements Accessible
public static class Box.Filler extends Component implements Accessible
public class BoxLayout extends Object implements LayoutManager2, Serializable
public class ButtonGroup extends Object implements Serializable
public class CellRendererPane extends Container implements Accessible
public class DebugGraphics extends Graphics
public class DefaultBoundedRangeModel extends Object implements BoundedRangeModel,
Serializable
```

```
public class DefaultButtonModel extends Object implements ButtonModel,
Serializable
public class DefaultCellEditor extends Object implements TableCellEditor,
TreeCellEditor, Serializable
public class DefaultComboBoxModel extends AbstractListModel implements
MutableComboBoxModel, Serializable
public class DefaultDesktopManager extends Object implements DesktopManager,
Serializable
public class DefaultFocusManager extends FocusManager
public class DefaultListCellRenderer extends JLabel implements ListCellRenderer,
Serializable
public static class DefaultListCellRenderer.UIResource extends
DefaultListCellRenderer implements UIResource
public class DefaultListModel extends AbstractListModel
public class DefaultListSelectionModel extends Object implements
ListSelectionModel, Cloneable, Serializable
public class DefaultSingleSelectionModel extends Object implements
SingleSelectionModel, Serializable
public abstract class FocusManager extends Object
public class GrayFilter extends RGBImageFilter
public class ImageIcon extends Object implements Icon, Serializable
public class JApplet extends Applet implements Accessible, RootPaneContainer
public class JButton extends AbstractButton implements Accessible
public class JCheckBox extends JToggleButton implements Accessible
public class JCheckBoxMenuItem extends JMenuItem implements SwingConstants,
Accessible
public class JColorChooser extends JComponent implements Accessible
public class JComboBox extends JComponent implements ItemSelectable,
ListDataListener, ActionListener, Accessible
public abstract class JComponent extends Container implements Serializable
public class JDesktopPane extends JLayeredPane implements Accessible
public class JDialog extends Dialog implements WindowConstants, Accessible,
RootPaneContainer
public class JEditorPane extends JTextComponent
public class JFileChooser extends JComponent implements Accessible
public class JFrame extends Frame implements WindowConstants, Accessible,
RootPaneContainer
public class JInternalFrame extends JComponent implements Accessible,
WindowConstants, RootPaneContainer
public static class JInternalFrame.JDesktopIcon extends JComponent implements
Accessible
public class JLabel extends JComponent implements SwingConstants, Accessible
public class JLayeredPane extends JComponent implements Accessible
public class JList extends JComponent implements Scrollable, Accessible
public class JMenu extends JMenuItem implements Accessible, MenuElement
public class JMenuBar extends JComponent implements Accessible, MenuElement
public class JMenuItem extends AbstractButton implements Accessible, MenuElement
public class JOptionPane extends JComponent implements Accessible
public class JPanel extends JComponent implements Accessible
public class JPasswordField extends JTextField
public class JPopupMenu extends JComponent implements Accessible, MenuElement
```

```
public static class JPopupMenu.Separator extends JSeparator
public class JProgressBar extends JComponent implements SwingConstants, Accessible
public class JRadioButton extends JToggleButton implements Accessible
public class JRadioButtonMenuItem extends JMenuItem implements Accessible
public class JRootPane extends JComponent implements Accessible
public class JScrollBar extends JComponent implements Adjustable, Accessible
public class JScrollPane extends JComponent implements ScrollPaneConstants,
Accessible
public class JSeparator extends JComponent implements SwingConstants, Accessible
public class JSlider extends JComponent implements SwingConstants, Accessible
public class JSplitPane extends JComponent implements Accessible
public class JTabbedPane extends JComponent implements Serializable, Accessible,
SwingConstants
public class JTable extends JComponent implements TableModelListener, Scrollable,
TableColumnModelListener, ListSelectionListener, CellEditorListener, Accessible
public class JTextArea extends JTextComponent
public class JTextField extends JTextComponent implements SwingConstants
public class JTextPane extends JEditorPane
public class JToggleButton extends AbstractButton implements Accessible
public static class JToggleButton.ToggleButtonModel extends DefaultButtonModel
public class JToolBar extends JComponent implements SwingConstants, Accessible
public static class JToolBar.Separator extends JSeparator
public class JToolTip extends JComponent implements Accessible
public class JTree extends JComponent implements Scrollable, Accessible
public static class JTree.DynamicUtilTreeNode extends DefaultMutableTreeNode
protected static class JTree.EmptySelectionModel extends DefaultTreeSelectionModel
public class JViewport extends JComponent implements Accessible
public class JWindow extends Window implements Accessible, RootPaneContainer
public class KeyStroke extends Object implements Serializable
public abstract class LookAndFeel extends Object
public class MenuSelectionManager extends Object
public class OverlayLayout extends Object implements LayoutManager2, Serializable
public class ProgressMonitor extends Object
public class ProgressMonitorInputStream extends FilterInputStream
public class RepaintManager extends Object
public class ScrollPaneLayout extends Object implements LayoutManager,
ScrollPaneConstants, Serializable
public static class ScrollPaneLayout.UIResource extends ScrollPaneLayout
implements UIResource
public class SizeRequirements extends Object implements Serializable
public class SwingUtilities extends Object implements SwingConstants
public class Timer extends Object implements Serializable
public class ToolTipManager extends MouseAdapter implements MouseMotionListener
public class UIDefaults extends Hashtable
public class UIManager extends Object implements Serializable
public static class UIManager.LookAndFeelInfo extends Object
public class ViewportLayout extends Object implements LayoutManager, Serializable
```

Enthaltene Schnittstellen

```
public abstract interface Action extends ActionListener
public abstract interface BoundedRangeModel
public abstract interface ButtonModel extends ItemSelectable
public abstract interface CellEditor
public abstract interface ComboBoxEditor
public abstract interface ComboBoxModel extends ListModel
public abstract interface DesktopManager
public abstract interface Icon
public abstract static interface JComboBox.KeySelectionManager
public abstract interface ListCellRenderer
public abstract interface ListModel
public abstract interface ListSelectionModel
public abstract interface MenuElement
public abstract interface MutableComboBoxModel extends ComboBoxModel
public abstract interface Renderer
public abstract interface RootPaneContainer
public abstract interface Scrollable
public abstract interface ScrollPaneConstants
public abstract interface SingleSelectionModel
public abstract interface SwingConstants
public abstract static interface UIDefaults.ActiveValue
public abstract static interface UIDefaults.LazyValue
public abstract interface WindowConstants
```

Enthaltene Ausnahmen

```
public class UnsupportedLookAndFeelException extends Exception
```

Verwandte Pakete

Das Konzept umfasst neben dem Hauptpaket `javax.swing` die folgenden Unterpakete:

```
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.plaf.multi
javax.swing.table
javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo
```

javax.swing.border

Ein Teilpaket für das Swing-Konzept.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.border` stellt Klassen und Schnittstellen zum Zeichnen von Rändern um eine Swing-Komponente herum bereit.

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. Das Paket `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes (so auch dieses) sind darunter angeordnet. Das Paket `javax.swing.border` erlaubt als konkrete Erweiterung der Basisfunktionalität das Zeichnen von vielfältigen Formen von Rändern um eine Swing-Komponente herum.

JDK-Version

Seit 1.2

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform in einen anderen Namensraum verlagert (in die `javax`-Struktur).

Enthaltene Klassen

```
public abstract class AbstractBorder extends Object implements Border,
Serializable
public class BevelBorder extends AbstractBorder
public class CompoundBorder extends AbstractBorder
public class EmptyBorder extends AbstractBorder implements Serializable
public class EtchedBorder extends AbstractBorder
public class LineBorder extends AbstractBorder
public class MatteBorder extends EmptyBorder
public class SoftBevelBorder extends BevelBorder
public class TitledBorder extends AbstractBorder
```

Enthaltene Schnittstellen

```
public abstract interface Border
```

Enthaltene Ausnahmen

(keine)

Verwandte Pakete

```
javax.swing
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.plaf.multi
javax.swing.table
javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo
```

javax.swing.colorchooser

Ein Teilpaket für das Swing-Konzept.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.colorchooser` stellt Klassen und Schnittstellen bereit, die von `JColorChooser`-Komponenten verwendet werden.

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet. Das Paket `javax.swing.colorchooser` ist eine Ergänzung der Funktionalität von `JColorChooser`-Komponenten bei Farboperationen.

JDK-Version

Seit 1.2

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert (in die `javax`-Struktur).

Enthaltene Klassen

```
public abstract class AbstractColorChooserPanel extends JPanel
public class ColorChooserComponentFactory extends Object
public class DefaultColorSelectionModel extends Object implements
ColorSelectionModel, Serializable
```

Enthaltene Schnittstellen

```
public abstract interface ColorSelectionModel
```

Enthaltene Ausnahmen

(keine)

Verwandte Pakete

```
javax.swing
javax.swing.border
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.plaf.multi
javax.swing.table
javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo
```

javax.swing.event

Ein Teilpaket für das Swing-Konzept, speziell für das Eventhandling unter Swing.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.event` stellt ergänzende Klassen und Schnittstellen für das Eventhandling unter dem Swingkonzept bereit.

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. Das Paket `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet. Das Paket `javax.swing.event` ist eine Ergänzung der Funktionalität im Rahmen des Java-Ereignismodells für solche Events, welche speziell von Swing-Komponenten erzeugt werden. Das Paket beinhaltet erweiternde Eventklassen und damit korrespondierende Eventlistener-Schnittstellen als Ergänzung der Events in dem Standard-Paket `java.awt.event` zum Reagieren auf Ereignisse im Allgemeinen.

JDK-Version

Seit 1.2

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert. Wurde das Swing-Konzept in den Vorgängerversionen des JDK 1.2-Final noch unter dem Namensraum `java.awt.swing` geführt, ist es jetzt – wie zahlreiche damit verbundene Pakete – in der `javax`-Struktur eingeordnet.

Enthaltene Klassen

```
public class AncestorEvent extends AWTEvent
public abstract class CaretEvent extends EventObject
public class ChangeEvent extends EventObject
public static final class DocumentEvent.EventType extends Object
public class EventListenerList extends Object implements Serializable
public class HyperlinkEvent extends EventObject
public static final class HyperlinkEvent.EventType extends Object
public abstract class InternalFrameAdapter extends Object implements
InternalFrameListener
public class InternalFrameEvent extends AWTEvent
```

```
public class ListDataEvent extends EventObject
public class ListSelectionEvent extends EventObject
public class MenuDragMouseEvent extends MouseEvent
public class MenuEvent extends EventObject
public class MenuKeyEvent extends KeyEvent
public abstract class MouseInputAdapter extends Object implements
MouseInputListener
public class PopupMenuEvent extends EventObject
public final class SwingPropertyChangeSupport extends PropertyChangeSupport
public class TableColumnModelEvent extends EventObject
public class TableModelEvent extends EventObject
public class TreeExpansionEvent extends EventObject
public class TreeModelEvent extends EventObject
public class TreeSelectionEvent extends EventObject
public class UndoableEditEvent extends EventObject
```

Enthaltene Schnittstellen

```
public abstract interface AncestorListener extends EventListener
public abstract interface CaretListener extends EventListener
public abstract interface CellEditorListener extends EventListener
public abstract interface ChangeListener extends EventListener
public abstract interface DocumentEvent
public abstract static interface DocumentEvent.ElementChange
public abstract interface DocumentListener extends EventListener
public abstract interface HyperlinkListener extends EventListener
public abstract interface InternalFrameListener extends EventListener
public abstract interface ListDataListener extends EventListener
public abstract interface ListSelectionListener extends EventListener
public abstract interface MenuDragMouseListener extends EventListener
public abstract interface MenuKeyListener extends EventListener
public abstract interface MenuListener extends EventListener
public abstract interface MouseInputListener extends MouseListener,
MouseMotionListener
public abstract interface PopupMenuListener extends EventListener
public abstract interface TableColumnModelListener extends EventListener
public abstract interface TableModelListener extends EventListener
public abstract interface TreeExpansionListener extends EventListener
public abstract interface TreeModelListener extends EventListener
public abstract interface TreeSelectionListener extends EventListener
public abstract interface TreeWillExpandListener extends EventListener
public abstract interface UndoableEditListener extends EventListener
```

Enthaltene Ausnahmen

(keine)

Verwandte Pakete

javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.plaf.multi
javax.swing.table
javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo

javax.swing.filechooser

Ein Teilpaket für das Swing-Konzept.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.filechooser` stellt Klassen und Schnittstellen bereit, die von `JFileChooser`-Komponenten verwendet werden.

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet. Das Paket `javax.swing.filechooser` ist eine spezielle Ergänzung der Funktionalität von `JFileChooser`-Komponenten.

JDK-Version

Seit 1.2

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert (in die `javax`-Struktur).

Enthaltene Klassen

```
public abstract class FileFilter extends Object
public abstract class FileSystemView extends Object
public abstract class FileView extends Object
```

Enthaltene Schnittstellen

(keine)

Enthaltene Ausnahmen

(keine)

Verwandte Pakete

javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.plaf.multi
javax.swing.table
javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo

javax.swing.plaf

Ein Teilpaket für das Swing-Konzept.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.plaf` stellt zahlreiche – oft abstrakte – Klassen und eine Schnittstelle bereit, die Swing zur Unterstützung seiner Look-and-Feel-Fähigkeiten benutzt.

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. Das Paket `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet. Das Paket `javax.swing.plaf` unterstützt die Look-and-Feel-Fähigkeit von Swing, also die Möglichkeiten von Java und Swing, dass sich das Aussehen und die Bedienung einer Benutzerschnittstelle den Spezifika einer speziellen Plattform anpassen.

JDK-Version

Seit 1.2

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert. Wurde das Swing-Konzept in den Vorgängerversionen des JDK 1.2-Final noch unter dem Namensraum `java.awt.swing` geführt, ist es jetzt – wie zahlreiche damit verbundene Pakete – in der `javax`-Struktur eingeordnet.

Enthaltene Klassen

```
public class BorderUIResource extends Object implements Border, UIResource,
Serializable
public static class BorderUIResource.BevelBorderUIResource extends BevelBorder
implements UIResource
public static class BorderUIResource.CompoundBorderUIResource extends
CompoundBorder implements UIResource
public static class BorderUIResource.EmptyBorderUIResource extends EmptyBorder
implements UIResource
public static class BorderUIResource.EtchedBorderUIResource extends EtchedBorder
implements UIResource
```

```
public static class BorderUIResource.LineBorderUIResource extends LineBorder
implements UIResource
public static class BorderUIResource.MatteBorderUIResource extends MatteBorder
implements UIResource
public static class BorderUIResource.TitledBorderUIResource extends TitledBorder
implements UIResource
public abstract class ButtonUI extends ComponentUI
public abstract class ColorChooserUI extends ComponentUI
public class ColorUIResource extends Color implements UIResource
public abstract class ComboBoxUI extends ComponentUI
public abstract class ComponentUI extends Object
public abstract class DesktopIconUI extends ComponentUI
public abstract class DesktopPaneUI extends ComponentUI
public class DimensionUIResource extends Dimension implements UIResource
public abstract class FileChooserUI extends ComponentUI
public class FontUIResource extends Font implements UIResource
public class IconUIResource extends Object implements Icon, UIResource,
Serializable
public class InsetsUIResource extends Insets implements UIResource
public abstract class InternalFrameUI extends ComponentUI
public abstract class LabelUI extends ComponentUI
public abstract class ListUI extends ComponentUI
public abstract class MenuBarUI extends ComponentUI
public abstract class MenuItemUI extends ButtonUI
public abstract class OptionPaneUI extends ComponentUI
public abstract class PanelUI extends ComponentUI
public abstract class PopupMenuUI extends ComponentUI
public abstract class ProgressBarUI extends ComponentUI
public abstract class ScrollBarUI extends ComponentUI
public abstract class ScrollPaneUI extends ComponentUI
public abstract class SeparatorUI extends ComponentUI
public abstract class SliderUI extends ComponentUI
public abstract class SplitPaneUI extends ComponentUI
public abstract class TabbedPaneUI extends ComponentUI
public abstract class TableHeaderUI extends ComponentUI
public abstract class TableUI extends ComponentUI
public abstract class TextUI extends ComponentUI
public abstract class ToolBarUI extends ComponentUI
public abstract class ToolTipUI extends ComponentUI
public abstract class TreeUI extends ComponentUI
public abstract class ViewportUI extends ComponentUI
```

Enthaltene Schnittstellen

```
public abstract interface UIResource
```

Enthaltene Ausnahmen

(keine)

Verwandte Pakete

javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.plaf.multi
javax.swing.table
javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo

javax.swing.plaf.basic

Ein Teilpaket für das Swing-Konzept.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.plaf.basic` unterstützt Anwenderschnittstellenobjekte im Zusammenhang mit dem Basis-Look-and-Feel von Swing.

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet. Das Paket `javax.swing.plaf` unterstützt Anwenderschnittstellenobjekte im Zusammenhang mit dem Basis-Look-and-Feel. Das Basis-Look-and-Feel ist das defaultmäßige Look-and-Feel von Swing-Komponenten. Es beinhaltet Komponenten, Layoutmanager, Events, Eventlistener und -adapter. Eine wichtige Anwendung von dieser Klasse ist das Erstellen von neuen Subklassen davon, um ein an persönliche Vorstellungen angepasstes Look-and-Feel zu erzeugen.

JDK-Version

Seit 1.2

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert. Wurde das Swing-Konzept in den Vorgängerversionen des JDK 1.2-Final noch unter dem Namensraum `java.awt.swing` geführt, ist es jetzt – wie zahlreiche damit verbundene Pakete – in der `javax`-Struktur eingeordnet.

Enthaltene Klassen

```
public class BasicArrowButton extends JButton implements SwingConstants
public class BasicBorders extends Object
public static class BasicBorders.ButtonBorder extends AbstractBorder implements
UIResource
public static class BasicBorders.FieldBorder extends AbstractBorder implements
UIResource
public static class BasicBorders.MarginBorder extends AbstractBorder implements
UIResource
```

```
public static class BasicBorders.MenuBarBorder extends AbstractBorder implements
UIResource
public static class BasicBorders.RadioButtonBorder extends
BasicBorders.ButtonBorder
public static class BasicBorders.SplitPaneBorder extends Object implements Border,
UIResource
public static class BasicBorders.ToggleButtonBorder extends
BasicBorders.ButtonBorder
public class BasicButtonListener extends Object implements MouseListener,
MouseMotionListener, FocusListener, ChangeListener,PropertyChangeListener
public class BasicButtonUI extends ButtonUI
public class BasicCheckBoxMenuItemUI extends BasicMenuItemUI
public class BasicCheckBoxUI extends BasicRadioButtonUI
public class BasicColorChooserUI extends ColorChooserUI
public class BasicComboBoxEditor extends Object implements ComboBoxEditor,
FocusListener
public static class BasicComboBoxEditor.UIResource extends BasicComboBoxEditor
implements UIResource
public class BasicComboBoxRenderer extends JLabel implements ListCellRenderer,
Serializable
public static class BasicComboBoxRenderer.UIResource extends BasicComboBoxRenderer
implements UIResource
public class BasicComboBoxUI extends ComboBoxUI
public class BasicComboPopup extends JPopupMenu implements ComboPopup
public class BasicDesktopIconUI extends DesktopIconUI
public class BasicDesktopPaneUI extends DesktopPaneUI
public class BasicDirectoryModel extends AbstractListModel implements
PropertyChangeListener
public class BasicEditorPaneUI extends BasicTextUI
public class BasicFileChooserUI extends FileChooserUI
public class BasicGraphicsUtils extends Object
public class BasicIconFactory extends Object implements Serializable
public class BasicInternalFrameTitlePane extends JComponent
public class BasicInternalFrameUI extends InternalFrameUI
public class BasicLabelUI extends LabelUI implements PropertyChangeListener
public class BasicListUI extends ListUI
public abstract class BasicLookAndFeel extends LookAndFeel implements Serializable
public class BasicMenuBarUI extends MenuBarUI
public class BasicMenuItemUI extends MenuItemUI
public class BasicMenuUI extends BasicMenuItemUI
public class BasicOptionPaneUI extends OptionPaneUI
public static class BasicOptionPaneUI.ButtonAreaLayout extends Object implements
LayoutManager
public class BasicPanelUI extends PanelUI
public class BasicPasswordFieldUI extends BasicTextFieldUI
public class BasicPopupMenuSeparatorUI extends BasicSeparatorUI
public class BasicPopupMenuUI extends PopupMenuUI
public class BasicProgressBarUI extends ProgressBarUI
public class BasicRadioButtonMenuItemUI extends BasicMenuItemUI
public class BasicRadioButtonUI extends BasicToggleButtonUI
```

```
public class BasicScrollBarUI extends ScrollBarUI implements LayoutManager,
SwingConstants
public class BasicScrollPaneUI extends ScrollPaneUI implements ScrollPaneConstants
public class BasicSeparatorUI extends SeparatorUI
public class BasicSliderUI extends SliderUI
public class BasicSplitPaneDivider extends Container implements
PropertyChangeListener
public class BasicSplitPaneUI extends SplitPaneUI
public class BasicTabbedPaneUI extends TabbedPaneUI implements SwingConstants
public class BasicTableHeaderUI extends TableHeaderUI
public class BasicTableUI extends TableUI
public class BasicTextAreaUI extends BasicTextUI
public class BasicTextFieldUI extends BasicTextUI
public class BasicTextPaneUI extends BasicEditorPaneUI
public abstract class BasicTextUI extends TextUI implements ViewFactory
public static class BasicTextUI.BasicCaret extends DefaultCaret implements
UIResource
public static class BasicTextUI.BasicHighlighter extends DefaultHighlighter
implements UIResource
public class BasicToggleButtonUI extends BasicButtonUI
public class BasicToolBarSeparatorUI extends BasicSeparatorUI
public class BasicToolBarUI extends ToolBarUI implements SwingConstants
public class BasicToolTipUI extends ToolTipUI
public class BasicTreeUI extends TreeUI
public class BasicViewportUI extends ViewportUI
public class DefaultMenuLayout extends BoxLayout implements UIResource
```

Enthaltene Schnittstellen

```
public abstract interface ComboPopup
```

Enthaltene Ausnahmen

(keine)

Verwandte Pakete

```
javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.metal
javax.swing.plaf.multi
javax.swing.table
javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo
```

javax.swing.plaf.metal

Ein Teilpaket für das Swing-Konzept.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.plaf.metal` unterstützt ein metallartiges Look-and-Feel von Swing-Elementen.

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet. Das Paket `javax.swing.plaf.metal` unterstützt ein metallartiges Aussehen von Swing-Elementen.

JDK-Version

Seit 1.2

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert (in die `javax`-Struktur).

Enthaltene Klassen

```
public class DefaultMetalTheme extends MetalTheme
public class MetalBorders extends Object
public static class MetalBorders.ButtonBorder extends AbstractBorder implements
UIResource
public static class MetalBorders.Flush3DBorder extends AbstractBorder implements
UIResource
public static class MetalBorders.InternalFrameBorder extends AbstractBorder
implements UIResource
public static class MetalBorders.MenuBarBorder extends AbstractBorder implements
UIResource
public static class MetalBorders.MenuItemBorder extends AbstractBorder implements
UIResource
public static class MetalBorders.PopupMenuBorder extends AbstractBorder implements
UIResource
public static class MetalBorders.RolloverButtonBorder extends
MetalBorders.ButtonBorder
```

```
public static class MetalBorders.ScrollPaneBorder extends AbstractBorder
implements UIResource
public static class MetalBorders.TextFieldBorder extends
MetalBorders.Flush3DBorder
public static class MetalBorders.ToolBarBorder extends AbstractBorder implements
UIResource, SwingConstants
public class MetalButtonUI extends BasicButtonUI
public class MetalCheckBoxIcon extends Object implements Icon, UIResource,
Serializable
public class MetalCheckBoxUI extends MetalRadioButtonUI
public class MetalComboBoxButton extends JButton
public class MetalComboBoxEditor extends BasicComboBoxEditor
public static class MetalComboBoxEditor.UIResource extends MetalComboBoxEditor
implements UIResource
public class MetalComboBoxIcon extends Object implements Icon, Serializable
public class MetalComboBoxUI extends BasicComboBoxUI
public class MetalDesktopIconUI extends BasicDesktopIconUI
public class MetalFileChooserUI extends BasicFileChooserUI
public class MetalIconFactory extends Object implements Serializable
public static class MetalIconFactory.FileIcon16 extends Object implements Icon,
Serializable
public static class MetalIconFactory.FolderIcon16 extends Object implements Icon,
Serializable
public static class MetalIconFactory.TreeControlIcon extends Object implements
Icon, Serializable
public static class MetalIconFactory.TreeFolderIcon extends
MetalIconFactory.FolderIcon16
public static class MetalIconFactory.TreeLeafIcon extends
MetalIconFactory.FileIcon16
public class MetalInternalFrameUI extends BasicInternalFrameUI
public class MetalLabelUI extends BasicLabelUI
public class MetalLookAndFeel extends BasicLookAndFeel
public class MetalPopupMenuSeparatorUI extends MetalSeparatorUI
public class MetalProgressBarUI extends BasicProgressBarUI
public class MetalRadioButtonUI extends BasicRadioButtonUI
public class MetalScrollBarUI extends BasicScrollBarUI
public class MetalScrollButton extends BasicArrowButton
public class MetalScrollPaneUI extends BasicScrollPaneUI
public class MetalSeparatorUI extends BasicSeparatorUI
public class MetalSliderUI extends BasicSliderUI
public class MetalSplitPaneUI extends BasicSplitPaneUI
public class MetalTabbedPaneUI extends BasicTabbedPaneUI
public class MetalTextFieldUI extends BasicTextFieldUI
public abstract class MetalTheme extends Object
public class MetalToggleButtonUI extends BasicToggleButtonUI
public class MetalToolBarUI extends BasicToolBarUI
public class MetalToolTipUI extends BasicToolTipUI
public class MetalTreeUI extends BasicTreeUI
```

Enthaltene Schnittstellen

(keine)

Enthaltene Ausnahmen

(keine)

Verwandte Pakete

```
javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.multi
javax.swing.table
javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo
```

javax.swing.plaf.multi

Ein Teilpaket für das Swing-Konzept.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.plaf.multi` unterstützt ein multiplexes Look-and-Feel von Swing-Elementen.

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet. Das Paket `javax.swing.plaf.multi` unterstützt ein multiplexes Look-and-Feel von Swing-Elementen.

JDK-Version

Seit 1.2

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert. Wurde das Swing-Konzept in den Vorgängerversionen des JDK 1.2-Final noch unter dem Namensraum `java.awt.swing` geführt, ist es jetzt – wie zahlreiche damit verbundene Pakete – in der `javax`-Struktur eingeordnet.

Enthaltene Klassen

```
public class MultiButtonUI extends ButtonUI
public class MultiColorChooserUI extends ColorChooserUI
public class MultiComboBoxUI extends ComboBoxUI
public class MultiDesktopIconUI extends DesktopIconUI
public class MultiDesktopPaneUI extends DesktopPaneUI
public class MultiFileChooserUI extends FileChooserUI
public class MultiInternalFrameUI extends InternalFrameUI
public class MultiLabelUI extends LabelUI
public class MultiListUI extends ListUI
public class MultiLookAndFeel extends LookAndFeel
public class MultiMenuBarUI extends MenuBarUI
public class MultiMenuItemUI extends MenuItemUI
public class MultiOptionPaneUI extends OptionPaneUI
```

```
public class MultiPanelUI extends PanelUI
public class MultiPopupMenuUI extends PopupMenuUI
public class MultiProgressBarUI extends ProgressBarUI
public class MultiScrollBarUI extends ScrollBarUI
public class MultiScrollPaneUI extends ScrollPaneUI
public class MultiSeparatorUI extends SeparatorUI
public class MultiSliderUI extends SliderUI
public class MultiSplitPaneUI extends SplitPaneUI
public class MultiTabbedPaneUI extends TabbedPaneUI
public class MultiTableHeaderUI extends TableHeaderUI
public class MultiTableUI extends TableUI
public class MultiTextUI extends TextUI
public class MultiToolBarUI extends ToolBarUI
public class MultiToolTipUI extends ToolTipUI
public class MultiTreeUI extends TreeUI
public class MultiViewportUI extends ViewportUI
```

Enthaltene Schnittstellen

(keine)

Enthaltene Ausnahmen

(keine)

Verwandte Pakete

```
javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.table
javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo
```

javax.swing.table

Ein Teilpaket für das Swing-Konzept.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.table` unterstützt das Paket `java.awt.swing.JTable`.

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet. Das Paket `javax.swing.table` beinhaltet Klassen und Interfaces für die Arbeit mit `java.awt.swing.JTable`.

JDK-Version

Seit 1.2

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert. Wurde das Swing-Konzept in den Vorgängerversionen des JDK 1.2-Final noch unter dem Namensraum `java.awt.swing` geführt, ist es jetzt – wie zahlreiche damit verbundene Pakete – in der `javax`-Struktur eingeordnet.

Enthaltene Klassen

```
public abstract class AbstractTableModel extends Object implements TableModel,
Serializable
public class DefaultTableCellRenderer extends JLabel implements TableCellRenderer,
Serializable
public static class DefaultTableCellRenderer.UIResource extends
DefaultTableCellRenderer implements UIResource
public class DefaultTableColumnModel extends Object implements TableColumnModel,
PropertyChangeListener, ListSelectionListener, Serializable
public class DefaultTableModel extends AbstractTableModel implements Serializable
public class JTableHeader extends JComponent implements TableColumnModelListener,
Accessible
public class TableColumn extends Object implements Serializable
```

Enthaltene Schnittstellen

```
public abstract interface TableCellEditor extends CellEditor
public abstract interface TableCellRenderer
public abstract interface TableColumnModel
public abstract interface TableModel
```

Enthaltene Ausnahmen

(keine)

Verwandte Pakete

```
javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.multi
javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo
```

javax.swing.text

Ein Teilpaket für das Swing-Konzept.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.text` beinhaltet Klassen und Interfaces für das Zusammenspiel mit editierbaren und nichteditierbaren Textkomponenten.

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet. Das Paket `javax.swing.text` beinhaltet Klassen und Interfaces für das Zusammenspiel mit editierbaren und nichteditierbaren Textkomponenten wie Textfelder und Textbereichen, Passwortfeldern oder Dokumenteneditoren. Die unterstützten Features sind beispielsweise die Selektion oder das Highlighting einer Komponente oder die Veränderung des Schriftstils.

JDK-Version

Seit 1.2

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert (in die `javax`-Struktur).

Die Klasse `DefaultTextUI` ist mittlerweile deprecated.

Enthaltene Klassen

```
public abstract class AbstractDocument extends Object implements Document,
Serializable
public static class AbstractDocument.ElementEdit extends AbstractUndoableEdit
implements DocumentEvent.ElementChange
public abstract class AbstractWriter extends Object
public class BoxView extends CompositeView
public class ComponentView extends View
public abstract class CompositeView extends View
public class DefaultCaret extends Rectangle implements Caret, FocusListener,
MouseListener, MouseMotionListener
public class DefaultEditorKit extends EditorKit
```

```
public static class DefaultEditorKit.BeepAction extends TextAction
public static class DefaultEditorKit.CopyAction extends TextAction
public static class DefaultEditorKit.CutAction extends TextAction
public static class DefaultEditorKit.DefaultKeyTypedAction extends TextAction
public static class DefaultEditorKit.InsertBreakAction extends TextAction
public static class DefaultEditorKit.InsertContentAction extends TextAction
public static class DefaultEditorKit.InsertTabAction extends TextAction
public static class DefaultEditorKit.PasteAction extends TextAction
public class DefaultHighlighter extends LayeredHighlighter
public static class DefaultHighlighter.DefaultHighlightPainter extends
LayeredHighlighter.LayerPainter
public class DefaultStyledDocument extends AbstractDocument implements
StyledDocument
public static class DefaultStyledDocument.AttributeUndoableEdit extends
AbstractUndoableEdit
public static class DefaultStyledDocument.ElementSpec extends Object
public abstract class DefaultTextUI extends BasicTextUI
public abstract class EditorKit extends Object implements Cloneable, Serializable
public class ElementIterator extends Object implements Cloneable
public class FieldView extends PlainView
public class GapContent extends javax.swing.text.GapVector implements
AbstractDocument.Content, Serializable
public class IconView extends View
public abstract class JTextComponent extends JComponent implements Scrollable,
Accessible
public static class JTextComponent.KeyBinding extends Object
public class LabelView extends View
public abstract class LayeredHighlighter extends Object implements Highlighter
public abstract static class LayeredHighlighter.LayerPainter extends Object
implements Highlighter.HighlightPainter
public class ParagraphView extends BoxView implements TabExpander
public class PasswordView extends FieldView
public class PlainDocument extends AbstractDocument
public class PlainView extends View implements TabExpander
public static final class Position.Bias extends Object
public class Segment extends Object
public class SimpleAttributeSet extends Object implements MutableAttributeSet,
Serializable, Cloneable
public final class StringContent extends Object implements
AbstractDocument.Content, Serializable
public class StyleConstants extends Object
public static class StyleConstants.CharacterConstants extends StyleConstants
implements AttributeSet.CharacterAttribute
public static class StyleConstants.ColorConstants extends StyleConstants
implements AttributeSet.ColorAttribute, AttributeSet.CharacterAttribute
public static class StyleConstants.FontConstants extends StyleConstants implements
AttributeSet.FontAttribute, AttributeSet.CharacterAttribute
public static class StyleConstants.ParagraphConstants extends StyleConstants
implements AttributeSet.ParagraphAttribute
public class StyleContext extends Object implements Serializable,
AbstractDocument.AttributeContext
```

```
public class StyledEditorKit extends DefaultEditorKit
public static class StyledEditorKit.AlignmentAction extends
StyledEditorKit.StyledTextAction
public static class StyledEditorKit.BoldAction extends
StyledEditorKit.StyledTextAction
public static class StyledEditorKit.FontFamilyAction extends
StyledEditorKit.StyledTextAction
public static class StyledEditorKit.FontSizeAction extends
StyledEditorKit.StyledTextAction
public static class StyledEditorKit.ForegroundAction extends
StyledEditorKit.StyledTextAction
public static class StyledEditorKit.ItalicAction extends
StyledEditorKit.StyledTextAction
public abstract static class StyledEditorKit.StyledTextAction extends TextAction
public static class StyledEditorKit.UnderlineAction extends
StyledEditorKit.StyledTextAction
public abstract class TableView extends BoxView
public class TabSet extends Object implements Serializable
public class TabStop extends Object implements Serializable
public abstract class TextAction extends AbstractAction
public class Utilities extends Object
public abstract class View extends Object implements SwingConstants
public class WrappedPlainView extends BoxView implements TabExpander
```

Enthaltene Schnittstellen

```
public abstract static interface AbstractDocument.AttributeContext
public abstract static interface AbstractDocument.Content
public abstract interface AttributeSet
public abstract static interface AttributeSet.CharacterAttribute
public abstract static interface AttributeSet.ColorAttribute
public abstract static interface AttributeSet.FontAttribute
public abstract static interface AttributeSet.ParagraphAttribute
public abstract interface Caret
public abstract interface Document
public abstract interface Element
public abstract interface Highlighter
public abstract static interface Highlighter.Highlight
public abstract static interface Highlighter.HighlightPainter
public abstract interface Keymap
public abstract interface MutableAttributeSet extends AttributeSet
public abstract interface Position
public abstract interface Style extends MutableAttributeSet
public abstract interface StyledDocument extends Document
public abstract interface TabableView
public abstract interface TabExpander
public abstract interface ViewFactory
```

Enthaltene Ausnahmen

```
public class BadLocationException extends Exception
public class ChangedCharSetException extends IOException
```

Verwandte Pakete

```
javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.multi
javax.swing.table
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo
```

javax.swing.text.html

Ein Teilpaket für das Swing-Konzept.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.text.html` unterstützt die Klasse `HTMLEditorKit` und Klassen für die Erstellung eines HTML-Texteditors.

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet. Das Paket `javax.swing.text.html` beinhaltet Klassen und Interfaces als Ergänzung der Klasse `HTMLEditorKit` und für die Erstellung eines HTML-Texteditors mit Unterstützung der zentralen HTML-Funktionalitäten (inklusive ergänzender Style-Sheets).

JDK-Version

Seit 1.2

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert. Wurde das Swing-Konzept in den Vorgängerversionen des JDK 1.2-Final noch unter dem Namensraum `java.awt.swing` geführt, ist es jetzt – wie zahlreiche damit verbundene Pakete – in der `javax`-Struktur eingeordnet.

Enthaltene Klassen

```
public class BlockView extends BoxView
public class CSS extends Object
public static final class CSS.Attribute extends Object
public class FormView extends ComponentView implements ActionListener
public class HTML extends Object
public static final class HTML extends Object
public static class HTML.Tag extends Object
public static class HTML.UnknownTag extends HTML.Tag implements Serializable
public class HTMLDocument extends DefaultStyledDocument
public abstract static class HTMLDocument.Iterator extends Object
public class HTMLEditorKit extends StyledEditorKit
```

```
public static class HTMLEditorKit.HTMLFactory extends Object implements
ViewFactory
public abstract static class HTMLEditorKit.HTMLTextAction extends
StyledEditorKit.StyledTextAction
public static class HTMLEditorKit.InsertHTMLTextAction extends
HTMLEditorKit.HTMLTextAction
public static class HTMLEditorKit.LinkController extends MouseAdapter implements
Serializable
public abstract static class HTMLEditorKit.Parser extends Object
public static class HTMLEditorKit.ParserCallback extends Object
public class HTMLFrameHyperlinkEvent extends HyperlinkEvent
public class HTMLWriter extends AbstractWriter
public class InlineView extends LabelView
public class ListView extends BlockView
public class MinimalHTMLWriter extends AbstractWriter
public class ObjectView extends ComponentView
public class Option extends Object
public class ParagraphView extends ParagraphView
public class StyleSheet extends StyleContext
public static class StyleSheet.BoxPainter extends Object implements Serializable
public static class StyleSheet.ListPainter extends Object implements Serializable
```

Enthaltene Schnittstellen

(keine)

Enthaltene Ausnahmen

(keine)

Verwandte Pakete

```
javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.multi
javax.swing.table
javax.swing.text
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo
```

javax.swing.text.html.parser

Ein Teilpaket für das Swing-Konzept.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.text.html.parser` unterstützt die Klasse `javax.swing.text.html` in Bezug auf Parser und die DTD-Deklaration (eine spezielle Vereinbarungsvorschrift unter HTML, die offiziell am Beginn jeder HTML-Datei stehen sollte und die Version von HTML spezifiziert).

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet. Das Paket `javax.swing.text.html.parser` unterstützt die Klasse `javax.swing.text.html` in Bezug auf Parser, SGML und die DTD-Deklaration per `DOCTPYE`. Im Sinn der strengen HTML-Syntax ist die erste Steueranweisung in der Webseite immer eine solche `DOCTYPE`-Anweisung (beispielsweise `<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">`), mit der die HTML als sogenannte DTD-Deklaration von SGML definiert wird. DTD steht dabei für Document Type Definition, was übersetzt Dokumententypen-Definition bedeutet und alle Besonderheiten, die ein Dokument bezüglich der Struktur und des Erscheinungsbildes haben darf, beschreibt. In der Praxis unterbleiben diese DTD-Deklarationen jedoch in der Regel.

JDK-Version

Seit 1.2

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert. Wurde das Swing-Konzept in den Vorgängerversionen des JDK 1.2-Final noch unter dem Namensraum `java.awt.swing` geführt, ist es jetzt – wie zahlreiche damit verbundene Pakete – in der `javax`-Struktur eingeordnet.

Enthaltene Klassen

```
public final class AttributeList extends Object implements DTDConstants,
Serializable
public final class ContentModel extends Object implements Serializable
public class DocumentParser extends Parser
public class DTD extends Object implements DTDConstants
public final class Element extends Object implements DTDConstants, Serializable
public final class Entity extends Object implements DTDConstants
public class Parser extends Object implements DTDConstants
public class ParserDelegator extends HTMLEditorKit.Parser
public class TagElement extends Object
```

Enthaltene Schnittstellen

```
public abstract interface DTDConstants
```

Enthaltene Ausnahmen

(keine)

Verwandte Pakete

```
javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.multi
javax.swing.table
javax.swing.text
javax.swing.text.html
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo
```

javax.swing.text.rtf

Ein Teilpaket für das Swing-Konzept.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.text.rtf` stellt eine Klasse für die Erstellung eines RTF-Editors bereit.

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. Das Paket `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet. Das Paket `javax.swing.text.rtf` stellt eine Klasse für die Erstellung eines RTF-Editors (Rich Text Format – ein Standardformat für den Textaustausch zwischen verschiedenen Plattformen) bereit.

JDK-Version

Seit 1.2 in die offizielle Dokumentation aufgenommen, aber dort noch unklar dokumentiert[18].

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert. Wurde das Swing-Konzept in den Vorgängerversionen des JDK 1.2-Final noch unter dem Namensraum `java.awt.swing` geführt, ist es jetzt – wie zahlreiche damit verbundene Pakete – in der `javax`-Struktur eingeordnet.

Enthaltene Klassen

```
public class RTFEditorKit extends StyledEditorKit
```

Enthaltene Schnittstellen

(keine)

18. **Zumindest in der dem Buch zugrunde liegenden Version der Dokumentation. Dort findet sich der Eintrag TBC, was nach inoffiziellen Sun-Quellen für To Be Confirmed (noch zu tun) lauten soll (ohne Gewähr).**

Enthaltene Ausnahmen

(keine)

Verwandte Pakete

```
javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.multi
javax.swing.table
javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.tree
javax.swing.undo
```

javax.swing.tree

Ein Teilpaket für das Swing-Konzept.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.tree` stellt Klassen und Interfaces für das Zusammenspiel mit `java.awt.swing.JTree` bereit.

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet. Das Paket `javax.swing.tree` stellt Klassen und Interfaces für das Zusammenspiel mit `java.awt.swing.JTree` bereit. Damit können Baumstrukturen (etwa zum Darstellen von Datenträgerinhalten) beispielsweise bezüglich ihrer Konstruktion, der Art eines Updates bei Veränderungen oder der Art der Anzeige von Daten kontrolliert werden.

JDK-Version

Seit 1.2

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert. Wurde das Swing-Konzept in den Vorgängerversionen des JDK 1.2-Final noch unter dem Namensraum `java.awt.swing` geführt, ist es jetzt – wie zahlreiche damit verbundene Pakete – in der `javax`-Struktur eingeordnet.

Enthaltene Klassen

```
public abstract class AbstractLayoutCache extends Object implements RowMapper
public abstract static class AbstractLayoutCache.NodeDimensions extends Object
public class DefaultMutableTreeNode extends Object implements Cloneable,
MutableTreeNode, Serializable
public class DefaultTreeCellEditor extends Object implements ActionListener,
TreeCellEditor, TreeSelectionListener
public class DefaultTreeCellRenderer extends JLabel implements TreeCellRenderer
public class DefaultTreeModel extends Object implements Serializable, TreeModel
public class DefaultTreeSelectionModel extends Object implements Cloneable,
Serializable, TreeSelectionModel
```

```
public class FixedHeightLayoutCache extends AbstractLayoutCache
public class TreePath extends Object implements Serializable
public class VariableHeightLayoutCache extends AbstractLayoutCache
```

Enthaltene Schnittstellen

```
public abstract interface MutableTreeNode extends TreeNode
public abstract interface RowMapper
public abstract interface TreeCellEditor extends CellEditor
public abstract interface TreeCellRenderer
public abstract interface TreeModel
public abstract interface TreeNode
public abstract interface TreeSelectionModel
```

Enthaltene Ausnahmen

```
public class ExpandVetoException extends Exception
```

Verwandte Pakete

```
javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.multi
javax.swing.table
javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.undo
```

javax.swing.undo

Ein Teilpaket für das Swing-Konzept.

Beschreibung

Swing besteht aus einer ganzen Sammlung von Paketen mit speziellen Schwerpunkten. Das Paket `javax.swing.undo` stellt Möglichkeiten zur Unterstützung von undo/redo-Aktionen in Applikationen (etwa einem Texteditor) bereit.

Anwendung

Swing ist der Teil der Java Foundation Classes (JFC), welche einen neuen Satz von GUI-Komponenten (Graphical User Interface = grafische Benutzerschnittstelle) implementiert. Unter dem gesamten Swing-Konzept versteht man eine Sammlung von Paketen, die eine Gestaltung der Oberfläche von Java-Applikationen und -Applets erlauben. `javax.swing` ist das oberste Basispaket in der Swing-Hierarchie und alle anderen Pakete des Konzeptes sind darunter angeordnet. Das Paket `javax.swing.undo` stellt Möglichkeiten zur Unterstützung von undo/redo-Aktionen in Applikationen (etwa einem Texteditor) bereit. Dabei werden sämtliche Aspekte dieser Aktionen unterstützt (inklusive Rückmeldungen über die entsprechenden Ausnahmen, falls die Aktion nicht durchgeführt werden konnte).

JDK-Version

Seit 1.2

Warnungen

Das gesamte Swing-Konzept wurde unter der Java-2-Plattform samt und sonders in einen anderen Namensraum verlagert (in die `javax`-Struktur).

Enthaltene Klassen

```
public class AbstractUndoableEdit extends Object implements UndoableEdit,
Serializable
public class CompoundEdit extends AbstractUndoableEdit
public class StateEdit extends AbstractUndoableEdit
public class UndoableEditSupport extends Object
public class UndoManager extends CompoundEdit implements UndoableEditListener
```

Enthaltene Schnittstellen

```
public abstract interface StateEditable
public abstract interface UndoableEdit
```

Enthaltene Ausnahmen

```
public class CannotRedoException extends RuntimeException
public class CannotUndoException extends RuntimeException
```

Verwandte Pakete

```
javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.multi
javax.swing.table
javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.tree
```

org.omg.CORBA

Das Hauptpaket des CORBA-Konzeptes.

Beschreibung

Dieses Paket bildet die Grundlage für die Zusammenarbeit zwischen Java und CORBA.

Anwendung

Java kann mittels IDL (Interfaces Definition Language) eine Verbindung zu anderen Verteilungsplattformen aufbauen. Eine der wichtigsten Vertreter davon ist CORBA (Common Object Request Broker Architecture). CORBA liegt derzeit in der Version 2 vor und ist ein plattformunabhängiger Standard, der die Kommunikation von Objekten im Netzwerk definiert. Java 2.0 bietet vollständigen CORBA-Support, indem der CORBA-Standard mit Hilfe des CORBA-IDL-Compilers integriert wird. Das Paket `org.omg.CORBA` bietet Unterstützung des Mappings des OMG-CORBA-API in die Java-Programmiersprache inklusive der Klasse `ORB`, welche als vollständiger Object Request Broker (ORB) implementiert ist. Auch ist in diesem Paket mit der Ausnahme `SystemException` die Rootexception aller weiteren CORBA-Ausnahmen enthalten.

JDK-Version

Seit 1.2

Warnungen

Unter der Finalversion des JDK 1.2 haben sich in der Java-IDL zahlreiche Veränderungen ergeben, welche im Wesentlichen aufgrund der 2.3-OMG-Spezifikation notwendig waren. Unter anderem sind die Klassen `Principal` und `PrincipalHolder` unter CORBA 2.2 als deprecated erklärt worden.

Enthaltene Klassen

```
public abstract class Any extends Object implements IDLEntity
public final class AnyHolder extends Object implements Streamable
public final class BooleanHolder extends Object implements Streamable
public final class ByteHolder extends Object implements Streamable
public final class CharHolder extends Object implements Streamable
public class CompletionStatus extends Object implements IDLEntity
public abstract class Context extends Object
public abstract class ContextList extends Object
public class DefinitionKind extends Object implements IDLEntity
public final class DoubleHolder extends Object implements Streamable
public abstract class DynamicImplementation extends ObjectImpl
public abstract class Environment extends Object
```

```
public abstract class ExceptionList extends Object
public final class FixedHolder extends Object implements Streamable
public final class FloatHolder extends Object implements Streamable
public final class IntHolder extends Object implements Streamable
public final class LongHolder extends Object implements Streamable
public abstract class NamedValue extends Object
public final class NameValuePair extends Object implements IDLEntity
public abstract class NVList extends Object
public final class ObjectHolder extends Object implements Streamable
public abstract class Principal extends Object
public final class PrincipalHolder extends Object implements Streamable
public abstract class Request extends Object
public abstract class ServerRequest extends Object
public final class ServiceDetail extends Object implements IDLEntity
public class ServiceDetailHelper extends Object
public final class ServiceInformation extends Object implements IDLEntity
public class ServiceInformationHelper extends Object
public final class ServiceInformationHolder extends Object implements Streamable
public class SetOverrideType extends Object implements IDLEntity
public final class ShortHolder extends Object implements Streamable
public final class StringHolder extends Object implements Streamable
public final class StructMember extends Object implements IDLEntity
public class TCKind extends Object
public abstract class TypeCode extends Object implements IDLEntity
public final class TypeCodeHolder extends Object implements Streamable
public final class UnionMember extends Object implements IDLEntity
public final class ValueMember extends Object implements IDLEntity
```

Enthaltene Schnittstellen

```
public abstract interface ARG_IN
public abstract interface ARG_INOUT
public abstract interface ARG_OUT
public abstract interface BAD_POLICY
public abstract interface BAD_POLICY_TYPE
public abstract interface BAD_POLICY_VALUE
public abstract interface CTX_RESTRICT_SCOPE
public abstract interface Current extends Object
public abstract interface DomainManager extends Object
public abstract interface DynAny extends Object
public abstract interface DynArray extends Object, DynAny
public abstract interface DynEnum extends Object, DynAny
public abstract interface DynFixed extends Object, DynAny
public abstract interface DynSequence extends Object, DynAny
public abstract interface DynStruct extends Object, DynAny
public abstract interface DynUnion extends Object, DynAny
public abstract interface DynValue extends Object, DynAny
public abstract interface IDLType extends Object, IDLEntity, IRObject
public abstract interface IRObject extends Object, IDLEntity
public abstract interface Object
public abstract interface PRIVATE_MEMBER
```

```
public abstract interface PUBLIC_MEMBER
public abstract interface UNSUPPORTED_POLICY
public abstract interface UNSUPPORTED_POLICY_VALUE
public abstract interface VM_ABSTRACT
public abstract interface VM_CUSTOM
public abstract interface VM_NONE
public abstract interface VM_TRUNCATABLE
```

Enthaltene Ausnahmen

```
public final class BAD_CONTEXT extends SystemException
public final class BAD_INV_ORDER extends SystemException
public final class BAD_OPERATION extends SystemException
public final class BAD_PARAM extends SystemException
public final class BAD_TYPECODE extends SystemException
public final class Bounds extends UserException
public final class COMM_FAILURE extends SystemException
public final class DATA_CONVERSION extends SystemException
public final class FREE_MEM extends SystemException
public final class IMP_LIMIT extends SystemException
public final class INITIALIZE extends SystemException
public final class INTERNAL extends SystemException
public final class INTF_REPOS extends SystemException
public final class INV_FLAG extends SystemException
public final class INV_IDENT extends SystemException
public final class INV_OBJREF extends SystemException
public class INV_POLICY extends SystemException
public final class INVALID_TRANSACTION extends SystemException
public final class MARSHAL extends SystemException
public final class NO_IMPLEMENT extends SystemException
public final class NO_MEMORY extends SystemException
public final class NO_PERMISSION extends SystemException
public final class NO_RESOURCES extends SystemException
public final class NO_RESPONSE extends SystemException
public final class OBJ_ADAPTER extends SystemException
public final class OBJECT_NOT_EXIST extends SystemException
public final class PERSIST_STORE extends SystemException
public final class PolicyError extends UserException
public abstract class SystemException extends RuntimeException
public final class TRANSACTION_REQUIRED extends SystemException
public final class TRANSACTION_ROLLEDBACK extends SystemException
public final class TRANSIENT extends SystemException
public final class UNKNOWN extends SystemException
public final class UnknownUserException extends UserException
public abstract class UserException extends Exception implements IDLEntity
public class WrongTransaction extends UserException
```

Verwandte Pakete

org.omg.CORBA.DynAnyPackage
org.omg.CORBA.ORBPackage
org.omg.CORBA.portable
org.omg.CORBA.TypeCodePackage
org.omg.CosNaming
org.omg.CosNaming.NamingContextPackage

org.omg.CORBA.DynAnyPackage

Ein Teilpaket des CORBA-Konzeptes.

Beschreibung

Unterstützung von Exceptions in Zusammenhang mit dem `DynAny`-Interface und dem CORBA-Konzept.

Anwendung

Java kann mittels IDL (Interfaces Definition Language) eine Verbindung zu CORBA aufbauen. Die Grundlage von dem CORBA-Support unter Java ist das Paket `org.omg.CORBA`. Das Paket `org.omg.CORBA.DynAnyPackage` beinhaltet nur Ausnahmen und unterstützt Exceptions in Zusammenhang mit dem `DynAny`-Interface aus dem CORBA-Hauptpaket.

JDK-Version

Seit 1.2

Warnungen

Unter der Finalversion des JDK 1.2 haben sich in der Java-IDL zahlreiche Veränderungen ergeben, welche im Wesentlichen aufgrund der 2.3-OMG-Spezifikation notwendig waren.

Enthaltene Klassen

(keine)

Enthaltene Schnittstellen

(keine)

Enthaltene Ausnahmen

```
public final class Invalid extends UserException
public final class InvalidSeq extends UserException
public final class InvalidValue extends UserException
public final class TypeMismatch extends UserException
```

Verwandte Pakete

```
org.omg.CORBA
org.omg.CORBA.ORBPackage
org.omg.CORBA.portable
org.omg.CORBA.TypeCodePackage
org.omg.CosNaming
org.omg.CosNaming.NamingContextPackage
```

org.omg.CORBA.ORBPackage

Ein Teilpaket des CORBA-Konzeptes.

Beschreibung

Allgemeine Unterstützung von den Exceptions `InvalidName` und `InconsistentTypeCode`.

Anwendung

Java kann mittels IDL (Interfaces Definition Language) eine Verbindung zu CORBA aufbauen. Die Grundlage von dem CORBA-Support von Java ist das Paket `org.omg.CORBA`. Das Paket `org.omg.CORBA.ORBPackage` bietet Unterstützung von der Exception `InvalidName`, welche von der Methode `ORB.resolve_initial_references` ausgeworfen wird, und der Exception `InconsistentTypeCode`, welche von den verschiedenen dynamischen Erstellungsmethoden in der `ORB`-Klasse aus dem Paket `org.omg.CORBA` ausgeworfen wird.

JDK-Version

Seit 1.2

Warnungen

Unter der Finalversion des JDK 1.2 haben sich in der Java-IDL zahlreiche Veränderungen ergeben, welche im Wesentlichen aufgrund der 2.3-OMG-Spezifikation notwendig waren.

Enthaltene Klassen

(keine)

Enthaltene Schnittstellen

(keine)

Enthaltene Ausnahmen

```
public final class InconsistentTypeCode extends UserException
public class InvalidName extends UserException
```

Verwandte Pakete

```
org.omg.CORBA
org.omg.CORBA.DynAnyPackage
org.omg.CORBA.portable
org.omg.CORBA.TypeCodePackage
org.omg.CosNaming
org.omg.CosNaming.NamingContextPackage
```

org.omg.CORBA.portable

Ein Teilpaket des CORBA-Konzeptes.

Beschreibung

Ein portierbarer Layer, das heisst ein Satz von ORB-API.

Anwendung

Java kann mittels IDL (Interfaces Definition Language) eine Verbindung zu CORBA aufbauen. Die Grundlage von dem CORBA-Support von Java ist das Paket `org.omg.CORBA`. Das Paket `org.omg.CORBA.ORBPackage` beinhaltet einen portierbaren Layer. Dieser stellt einen Satz von ORB-APIs dar, welche es erlauben, dass

Code, welcher für einen Vendor generiert wurde, in gleicher Weise auf anderen Vendor- ORBs läuft.

JDK-Version

Seit 1.2

Warnungen

Unter der Finalversion des JDK 1.2 haben sich in der Java-IDL zahlreiche Veränderungen ergeben, welche im Wesentlichen aufgrund der 2.3-OMG-Spezifikation notwendig waren.

Enthaltene Klassen

```
public abstract class Delegate extends Object
public abstract class InputStream extends InputStream
public abstract class ObjectImpl extends Object implements Object
public abstract class OutputStream extends OutputStream
public class ServantObject extends Object
```

Enthaltene Schnittstellen

```
public abstract interface IDLEntity extends Serializable
public abstract interface InvokeHandler
public abstract interface ResponseHandler
public abstract interface Streamable
```

Enthaltene Ausnahmen

```
public class ApplicationException extends Exception
public final class RemarshalException extends Exception
```

Verwandte Pakete

```
org.omg.CORBA
org.omg.CORBA.DynAnyPackage
org.omg.CORBA.ORBPackage
org.omg.CORBA.TypeCodePackage
org.omg.CosNaming
org.omg.CosNaming.NamingContextPackage
```

org.omg.CORBA.TypeCodePackage

Ein Teilpaket des CORBA-Konzeptes.

Beschreibung

Unterstützung der Exceptions `BadKind` und `Bounds`.

Anwendung

Java kann mittels IDL (Interfaces Definition Language) eine Verbindung zu CORBA aufbauen. Die Grundlage von dem CORBA-Support von Java ist das Paket `org.omg.CORBA`. Das Paket `org.omg.CORBA.TypeCodePackage` beinhaltet Unterstützung der benutzerdefinierten Exceptions `BadKind` und `Bounds`, welche von Methoden in der Klasse `TypeCode` im CORBA-Hauptpaket ausgeworfen werden.

JDK-Version

Seit 1.2

Warnungen

Unter der Finalversion des JDK 1.2 haben sich in der Java-IDL zahlreiche Veränderungen ergeben, welche im Wesentlichen aufgrund der 2.3-OMG-Spezifikation notwendig waren.

Enthaltene Klassen

(keine)

Enthaltene Schnittstellen

(keine)

Enthaltene Ausnahmen

```
public final class BadKind extends UserException
public final class Bounds extends UserException
```

Verwandte Pakete

```
org.omg.CORBA
org.omg.CORBA.DynAnyPackage
org.omg.CORBA.ORBPackage
org.omg.CORBA.portable
org.omg.CosNaming
org.omg.CosNaming.NamingContextPackage
```

org.omg.CosNaming

Ein Teilpaket des CORBA-Konzeptes.

Beschreibung

Unterstützung des Namingservice für Java-IDL.

Anwendung

Java kann mittels IDL (Interfaces Definition Language) eine Verbindung zu CORBA aufbauen. Die Grundlage des CORBA-Supports von Java ist das Paket `org.omg.CORBA`. Das Paket `org.omg.CosNaming` beinhaltet Unterstützung von dem Namingservice für besagte Java-IDL.

JDK-Version

Seit 1.2

Warnungen

Unter der Finalversion des JDK 1.2 haben sich in der Java-IDL zahlreiche Veränderungen ergeben, welche im Wesentlichen aufgrund der 2.3-OMG-Spezifikation notwendig waren.

Enthaltene Klassen

```
public abstract class _BindingIteratorImplBase extends DynamicImplementation
implements BindingIterator
public class  extends ObjectImpl implements BindingIterator
public abstract class _NamingContextImplBase extends DynamicImplementation
implements NamingContext
public class _NamingContextStub extends ObjectImpl implements NamingContext
public final class Binding extends Object implements IDLEntity
public class BindingHelper extends Object public final class BindingHolder extends
Object implements Streamable
public class BindingIteratorHelper extends Object
public final class BindingIteratorHolder extends Object implements Streamable
public class BindingListHelper extends Object
public final class BindingListHolder extends Object implements Streamable
public final class BindingType extends Object implements IDLEntity
public class BindingTypeHelper extends Object
public final class BindingTypeHolder extends Object implements Streamable
public class IstringHelper extends Object
public final class NameComponent extends Object implements IDLEntity
public class NameComponentHelper extends Object
public final class NameComponentHolder extends Object implements Streamable
public class NameHelper extends Object
```

```
public final class NameHolder extends Object implements Streamable
public class NamingContextHelper extends Object
public final class NamingContextHolder extends Object implements Streamable
```

Enthaltene Schnittstellen

```
public abstract interface BindingIterator extends Object, IDLEntity
public abstract interface NamingContext extends Object, IDLEntity
```

Enthaltene Ausnahmen

(keine)

Verwandte Pakete

```
org.omg.CORBA
org.omg.CORBA.DynAnyPackage
org.omg.CORBA.ORBPackage
org.omg.CORBA.portable
org.omg.TypeCodePackage
org.omg.CosNaming.NamingContextPackage
```

org.omg.CosNaming.NamingContextPackage

Ein Teilpaket des CORBA-Konzeptes.

Beschreibung

Allgemeine Unterstützung des Pakets `org.omg.CosNaming`.

Anwendung

Java kann mittels IDL (Interfaces Definition Language) eine Verbindung zu CORBA aufbauen. Die Grundlage von dem CORBA-Support von Java ist das Paket `org.omg.CORBA`. Das Paket `org.omg.CosNaming.NamingContextPackage` beinhaltet Unterstützung der Ausnahmen, welche in dem Package `org.omg.CosNaming` verwendet werden (`AlreadyBound`, `CannotProceed`, `InvalidName`, `NotEmpty` und `NotFound`).

JDK-Version

Seit 1.2

Warnungen

Unter der Finalversion des JDK 1.2 haben sich in der Java-IDL zahlreiche Veränderungen ergeben, welche im Wesentlichen aufgrund der 2.3-OMG-Spezifikation notwendig waren.

Enthaltene Klassen

```
public class AlreadyBoundHelper extends Object
public final class AlreadyBoundHolder extends Object implements Streamable
public class CannotProceedHelper extends Object
public final class CannotProceedHolder extends Object implements Streamable
public class InvalidNameHelper extends Object
public final class InvalidNameHolder extends Object implements Streamable
public class NotEmptyHelper extends Object
public final class NotEmptyHolder extends Object implements Streamable
public class NotFoundHelper extends Object
public final class NotFoundHolder extends Object implements Streamable
public final class NotFoundReason extends Object implements IDLEntity
public class NotFoundReasonHelper extends Object
public final class NotFoundReasonHolder extends Object implements Streamable
```

Enthaltene Schnittstellen

(keine)

Enthaltene Ausnahmen

```
public final class AlreadyBound extends UserException implements IDLEntity
public final class CannotProceed extends UserException implements IDLEntity
public final class InvalidName extends UserException implements IDLEntity
public final class NotEmpty extends UserException implements IDLEntity
public final class NotFound extends UserException implements IDLEntity
```

Verwandte Pakete

```
org.omg.CORBA
org.omg.CORBA.DynAnyPackage
org.omg.CORBA.ORBPackage
org.omg.CORBA.portable
org.omg.TypeCodePackage
org.omg.CosNaming.CosNaming
```

Deprecated-erklärte Elemente des Java-API 1.2

Die nachfolgenden Schnittstellen, Klassen, Methoden, Konstruktoren und Variablen werden unter der Java-2-Plattform von Sun als veraltet beziehungsweise verworfen (deprecated) bezeichnet. Sie werden dennoch in dem API der Java-2-Plattform (dem 1.2-API) mitaufgenommen, um die Kompatibilität zu den vorangegangenen Versionen sicherzustellen. Es gibt jedoch für fast jedes der veralteten Elemente mittlerweile neuere Lösungen oder die Funktionalität wird nicht mehr benötigt. Die Verwendung dieser veralteten Elemente ist keinesfalls verboten (und in vielen Fällen sogar noch notwendig – beispielsweise für Applets, denn viele Browser unterstützten den 2.0-Standard oder gar den 1.1-Standard von Java noch nicht). Ein als »deprecated« gekennzeichnetes Element ist nicht mehr und nicht weniger als eine Klasse oder ein Klassenbestandteil von einer Vorgängerversion der aktuellen Java-Plattform, für die es in dem neuen API eine bessere Lösung oder ein geändertes Konzept gibt.

Um die Veränderungen in dem Java-API auf einen Blick zur Verfügung zu haben, folgt hier die Auflistung der als deprecated gekennzeichneten Elemente. Dabei werden in der Auflistung alle Klassen, Schnittstellen, Ausnahmen, Variablen, Methoden und Konstruktoren angegeben, sowie optional einige Informationen darüber, wann die Elemente als deprecated gekennzeichnet wurden und welcher Ersatz dafür verwendet werden kann. Detailliertere Informationen finden Sie in der API-Dokumentation.

Deprecated-Klassen

Klasse	Erklärung
javax.swing.text.DefaultTextUI	
java.security.Identity	Die Klasse wird nicht mehr in der 1.2-API verwendet. Die Funktionalität ist ersetzt durch java.security.KeyStore, das java.security.cert-Paket, und java.security.Principal.
java.security.IdentityScope	Die Klasse wird nicht mehr in der 1.2-API verwendet. Die Funktionalität ist ersetzt durch java.security.KeyStore, das java.security.cert-Paket, und java.security.Principal.
java.io.LineNumberInputStream	Diese Klasse war nicht ganz fehlerfrei. Verlagerung der Funktionalität seit dem JDK 1.1 im Wesentlichen in die Characterstream-Klassen.
java.rmi.server.LogStream	Keine direkte Entsprechung.
java.rmi.server.Operation	Keine direkte Entsprechung.
org.omg.CORBA.Principal	Deprecated durch CORBA 2.2.
org.omg.CORBA.PrincipalHolder	Deprecated durch CORBA 2.2.
java.security.Signer	Die Klasse wird nicht mehr in der 1.2-API verwendet. Die Funktionalität ist ersetzt durch java.security.KeyStore, das java.security.cert-Paket, und java.security.Principal.
java.io.StringBufferInputStream	Diese Klasse war nicht ganz fehlerfrei. Verlagerung der Funktionalität seit dem JDK 1.1 in die StringReader-Klasse.

Deprecated-Schnittstellen

Schnittstelle	Erklärung
`java.security.Certificate`	Ersetzt durch ein neues Certificate-handling-Paket im JDK1.2.
`java.rmi.server.LoaderHandler`	Keine direkte Entsprechung.
`java.rmi.registry.RegistryHandler`	Keine direkte Entsprechung.
`java.rmi.server.RemoteCall`	Keine direkte Entsprechung.
`java.rmi.server.Skeleton`	Keine direkte Entsprechung. Skeletons werden für Remote-Methodenaufrufe im JDK1.2 nicht mehr benötigt.

Deprecated-Ausnahmen

Ausnahme	Erklärung
`java.rmi.RMISecurityException`	Keine direkte Entsprechung.
`java.rmi.ServerRuntimeException`	Keine direkte Entsprechung.
`java.rmi.server.SkeletonMismatchException`	Keine direkte Entsprechung.
`java.rmi.server.SkeletonNotFoundException`	Keine direkte Entsprechung.

Deprecated-Variablen

Variable	Erklärung
java.awt.Frame.CROSSHAIR_CURSOR	Ersetzt durch Cursor.CROSSHAIR_CURSOR.
java.awt.Frame.DEFAULT_CURSOR	Ersetzt durch Cursor.DEFAULT_CURSOR.
java.awt.Frame.E_RESIZE_CURSOR	Ersetzt durch Cursor.E_RESIZE_CURSOR.
java.awt.Frame.HAND_CURSOR	Ersetzt durch Cursor.HAND_CURSOR.
java.lang.SecurityManager.inCheck	Diese Form der Sicherheitsüberprüfung ist nicht länger notwendig. Empfohlen wird statt dessen ein checkPermission-Aufruf.
java.awt.Frame.MOVE_CURSOR	Ersetzt durch Cursor.MOVE_CURSOR.
java.awt.Frame.N_RESIZE_CURSOR	Ersetzt durch Cursor.N_RESIZE_CURSOR.
java.awt.Frame.NE_RESIZE_CURSOR	Ersetzt durch Cursor.NE_RESIZE_CURSOR.
java.awt.Frame.NW_RESIZE_CURSOR	Ersetzt durch Cursor.NW_RESIZE_CURSOR.
java.awt.Frame.S_RESIZE_CURSOR	Ersetzt durch Cursor.S_RESIZE_CURSOR.
java.awt.Frame.SE_RESIZE_CURSOR	Ersetzt durch Cursor.SE_RESIZE_CURSOR.
java.awt.Frame.SW_RESIZE_CURSOR	Ersetzt durch Cursor.SW_RESIZE_CURSOR.
java.awt.Frame.TEXT_CURSOR	Ersetzt durch Cursor.TEXT_CURSOR.
java.awt.Frame.W_RESIZE_CURSOR	Ersetzt durch Cursor.W_RESIZE_CURSOR.
java.awt.Frame.WAIT_CURSOR	Ersetzt durch Cursor.WAIT_CURSOR.

Deprecated-Methoden

Methode	Erklärung	Ersetzt durch
`java.awt.Component.action (Event, Object)`	Das gesamte Event-Konzept ist bereits im JDK 1.1 umgestellt worden.	Die Funktionalität wird über `ActionListener` realisiert.
`java.awt.List.addItem(String)`		`add(String).`
`java.awt.List.addItem(String, int)`		`add(String, int).`
`java.awt.CardLayout.addLayout Component(String, Component)`		`addLayoutComponent (Component, Object).`
`java.awt.BorderLayout. addLayoutComponent(String, Component)`		`addLayoutComponent (Component, Object).`
`java.awt.List. allowsMultipleSelections()`	Bereits im JDK 1.1 abgelöst	`isMultipleMode().`
`java.lang.ThreadGroup. allowThreadSuspension (boolean)`	Die Definition von diesem Aufruf basiert auf `ThreadGroup. suspend()`. Diese Methode ist deprecated.	
`java.awt.TextArea. appendText(String)`	Bereits im JDK 1.1 abgelöst	`append(String).`
`java.awt.Component.bounds()`	Bereits im JDK 1.1 abgelöst	`getBounds().`
`java.lang.SecurityManager. classDepth(String)`	Die Sicherheits-überprüfung gehört zum veralteten Sicherheits-konzept von Java.	`checkPermission`
`java.lang.SecurityManager. classLoaderDepth()`	Die Sicherheits-überprüfung gehört zum veralteten Sicherheits-konzept von Java.	`checkPermission`
`java.awt.List.clear()`	Bereits im JDK 1.1 abgelöst	`removeAll().`
`java.awt.Container. countComponents()`	Bereits im JDK 1.1 abgelöst	`getComponentCount().`
`java.awt.Choice.countItems()`	Bereits im JDK 1.1 abgelöst	`getItemCount().`

Methode	Erklärung	Ersetzt durch
java.awt.Menu.countItems()	Bereits im JDK 1.1 abgelöst	getItemCount().
java.awt.List.countItems()	Bereits im JDK 1.1 abgelöst	getItemCount().
java.awt.MenuBar.countMenus()	Bereits im JDK 1.1 abgelöst	getMenuCount().
java.lang.Thread. countStackFrames()	Die Definition von diesem Aufruf basiert auf Thread. suspend(). Diese Methode ist deprecated.	
org.omg.CORBA.ORB. create_recursive_sequence_tc (int, int)		
javax.swing.JTable. createScrollPaneForTable (JTable)	Stammt noch aus der Swing-Version 1.0.2.	JScrollPane(aTable).
java.lang.SecurityManager. currentClassLoader()	Die Sicherheits-überprüfung gehört zum veralteten Sicherheitskonzept von Java.	checkPermission
java.lang.SecurityManager. currentLoadedClass()	Die Sicherheits-überprüfung gehört zum veralteten Sicherheitskonzept von Java.	checkPermission
java.lang.ClassLoader. defineClass(byte[], int, int)		defineClass (java.lang.String, byte[], int, int)
java.awt.List.delItem(int)		remove(String) und remove(int).
java.awt.List.delItems(int, int)	Bereits im JDK 1.1 abgelöst. Soll nur als private Methode des Paketes verwendet werden.	
java.awt.Component. deliverEvent(Event)	Bereits im JDK 1.1 abgelöst	dispatchEvent(AWTEvent e).
java.awt.Container. deliverEvent(Event)	Bereits im JDK 1.1 abgelöst	dispatchEvent(AWTEvent e).

Methode	Erklärung	Ersetzt durch
java.awt.Component.disable()	Bereits im JDK 1.1 abgelöst	setEnabled(boolean).
java.awt.MenuItem.disable()	Bereits im JDK 1.1 abgelöst	setEnabled(boolean).
java.rmi.server.Skeleton. dispatch(Remote, RemoteCall, int, long)	Keine direkte Entsprechung.	
java.rmi.server.RemoteCall. done()	Keine direkte Entsprechung.	
java.rmi.server.RemoteRef. done(RemoteCall)	JDK1.2 Stylestubs verwenden diese Methode nicht länger.	
java.awt.Component.enable()	Bereits im JDK 1.1 abgelöst	setEnabled(boolean).
java.awt.MenuItem.enable()	Bereits im JDK 1.1 abgelöst	setEnabled(boolean).
java.awt.Component.enable (boolean)	Bereits im JDK 1.1 abgelöst	setEnabled(boolean).
java.awt.MenuItem.enable (boolean)	Bereits im JDK 1.1 abgelöst	setEnabled(boolean).
java.security.SignatureSpi. engineGetParameter(String)		
java.security.SignatureSpi. engineSetParameter(String, Object)		engineSetParameter.()
org.omg.CORBA.ServerRequest. except(Any)		set_exception
java.rmi.server.RemoteCall. executeCall()	Keine direkte Entsprechung.	
org.omg.CORBA.Any. extract_Principal()	Deprecated durch CORBA 2.2.	
org.omg.CORBA.ORB. get_current()		resolve_initial_references
java.security.Security. getAlgorithmProperty(String, String)	Abgelöst durch Funktionalität im Sicherheitsmodell des JDK 1.2.	
java.sql.ResultSet. getBigDecimal(int, int)		
java.sql.CallableStatement. getBigDecimal(int, int)		

Methode	Erklärung	Ersetzt durch
java.sql.ResultSet. getBigDecimal(String, int)		
java.awt.Polygon. getBoundingBox()	Bereits im JDK 1.1 abgelöst	getBounds().
java.lang.String. getBytes(int, int, byte[], int)	Da unzuverlässig, soll diese Methode seit dem JDK 1.1 nicht mehr verwendet werden.	getBytes(String enc)
java.awt.Graphics. getClipRect()	Bereits im JDK 1.1 abgelöst	getClipBounds().
java.awt.CheckboxGroup. getCurrent()	Bereits im JDK 1.1 abgelöst	getSelectedCheckbox().
java.awt.Frame. getCursorType()	Bereits im JDK 1.1 abgelöst	Component.getCursor().
java.sql.Time.getDate()	Bereits im JDK 1.1 abgelöst	Calendar.get (Calendar.DAY_OF_MONTH).
java.util.Date.getDate()		
java.sql.Time.getDay()	Bereits im JDK 1.1 abgelöst	Calendar.get (Calendar.DAY_OF_WEEK).
java.util.Date.getDay()		
java.rmi.server.LogStream. getDefaultStream()	Keine direkte Entsprechung.	
java.lang.System.getenv (String)		java.lang.System.getProperty () und getTypeName()
java.awt.Toolkit. getFontList()	Neue Funktionalität unter Graphics-Environment.	
java.awt.Toolkit. getFontMetrics(Font)		
java.awt.Toolkit.getFontPeer (String, int)		
java.sql.Date.getHours()	Bereits im JDK 1.1 abgelöst	Calendar.get (Calendar.HOUR_OF_DAY).
java.util.Date.getHours()		
java.lang.SecurityManager. getInCheck()	Die Sicherheitsüberprüfung gehört zum veralteten Sicherheitskonzept von Java.	checkPermission

Methode	Erklärung	Ersetzt durch
java.rmi.server.RemoteCall. getInputStream()	Keine direkte Entsprechung.	
javax.swing.KeyStroke. getKeyStroke(char, boolean)		GETKEYSTROKE(CHAR)
javax.swing.AbstractButton. getLabel()		getText()
java.awt.Scrollbar. getLineIncrement()	Bereits im JDK 1.1 abgelöst	getUnitIncrement().
java.lang.Runtime. getLocalizedInputStream(InputStream)	Bereits im JDK 1.1 abgelöst	InputStreamReader und BufferedReader-Klassen
java.lang.Runtime. getLocalizedOutputStream (OutputStream)	Bereits im JDK 1.1 abgelöst	OutputStreamWriter, BufferedWriter, und PrintWriter-Klassen
java.sql.DriverManager. getLogStream()	Bereits im JDK 1.1 abgelöst	getMaxDescent().
java.awt.FontMetrics. getMaxDecent()		
javax.swing.JInternalFrame. getMenuBar()	Ursprünglich in der Swing-Version 1.0.3 verwendet. Das vollständige Swingkonzept wurde überarbeitet.	getJMenuBar().
javax.swing.JRootPane. getMenuBar()	Ursprünglich in der Swing-Version 1.0.3 verwendet. Das vollständige Swingkonzept wurde überarbeitet.	getJMenuBar().
java.sql.Date.getMinutes()	Bereits im JDK 1.1 abgelöst	Calendar.get (Calendar.MINUTE).
java.util.Date.getMinutes()		
java.sql.Time.getMonth()	Bereits im JDK 1.1 abgelöst	Calendar.get (Calendar.MONTH).
java.util.Date.getMonth()		
java.rmi.server.Operation. getOperation()	Keine direkte Entsprechung.	
java.rmi.server.Skeleton. getOperations()	Keine direkte Entsprechung.	

Methode	Erklärung	Ersetzt durch
`java.rmi.server.LogStream.getOutputStream()`	Keine direkte Entsprechung.	
`java.rmi.server.RemoteCall.getOutputStream()`	Keine direkte Entsprechung.	
`java.awt.Scrollbar.getPageIncrement()`	Bereits im JDK 1.1 abgelöst	`getBlockIncrement()`.
`java.security.Signature.getParameter(String)`		
`java.awt.Component.getPeer()`	Bereits im JDK 1.1 wurde eingeführt, dass Programme Peers nicht direkt manipulieren sollen.	`boolean isDisplayable()`.
`java.awt.MenuComponent.getPeer()`	Bereits im JDK 1.1 wurde eingeführt, dass Programme Peers nicht direkt manipulieren sollen.	
`java.awt.Font.getPeer()`	Fontrendering ist mittlerweile plattformunabhängig.	
`java.rmi.server.RemoteCall.getResultStream(boolean)`	Keine direkte Entsprechung.	
`java.sql.Date.getSeconds()`	Bereits im JDK 1.1 abgelöst	`Calendar.get(Calendar.SECOND)`.
`java.util.Date.getSeconds()`		
`java.rmi.server.LoaderHandler.getSecurityContext(ClassLoader)`	Keine direkte Entsprechung.	
`java.rmi.server.RMIClassLoader.getSecurityContext(ClassLoader)`	Keine direkte Entsprechung. Seit dem JDK1.2 verwendet RMI diese Methode nicht mehr.	
`javax.swing.JPasswordField.getText()`	In der JDK-Version 1.2 abgelöst	`getPassword()`.
`javax.swing.JPasswordField.getText(int, int)`	In der JDK-Version 1.2 abgelöst	`getPassword()`.

Methode	Erklärung	Ersetzt durch
java.util.Date. getTimezoneOffset()	Bereits im JDK 1.1 abgelöst	Calendar.get (Calendar.ZONE_OFFSET) + Calendar.get (Calendar.DST_OFFSET).
java.net.MulticastSocket. getTTL()		getTimeToLive()
java.net.DatagramSocketImpl. getTTL()		getTimeToLive()
java.sql.ResultSet. getUnicodeStream(int)		
java.sql.ResultSet. getUnicodeStream(String)		
javax.swing.ScrollPaneLayout. getViewportBorderBounds (JScrollPane)	Ursprünglich bis Swing-Version 1.1 verwendet. Das vollständige Swingkonzept wurde überarbeitet.	JScrollPane. getViewportBorderBounds().
java.awt.Scrollbar. getVisible()	Bereits im JDK 1.1 abgelöst	getVisibleAmount().
java.sql.Time.getYear()	Bereits im JDK 1.1 abgelöst	Calendar.get(Calendar.YEAR) - 1900.
java.util.Date.getYear()		
java.awt.Component. getFocus(Event, Object)	Bereits im JDK 1.1 abgelöst	processFocusEvent (FocusEvent).
java.awt.Component. handleEvent(Event)	Bereits im JDK 1.1 abgelöst	processEvent(AWTEvent).
java.awt.Component.hide()	Bereits im JDK 1.1 abgelöst	setVisible(boolean).
java.lang.SecurityManager. inClass(String)	Die Sicherheits-überprüfung gehört zum veralteten Sicherheits-konzept von Java.	checkPermission
java.lang.SecurityManager. inClassLoader()	Die Sicherheits-überprüfung gehört zum veralteten Sicherheits-konzept von Java.	checkPermission
org.omg.CORBA.Any. insert_Principal(Principal)	Deprecated durch CORBA 2.2.	

Methode	Erklärung	Ersetzt durch
java.awt.TextArea.insertText(String, int)	Bereits im JDK 1.1 abgelöst	insert(String, int).
java.awt.Container.insets()	Bereits im JDK 1.1 abgelöst	getInsets().
java.awt.Component.inside(int, int)	Bereits im JDK 1.1 abgelöst	contains(int, int).
java.awt.Polygon.inside(int, int)	Bereits im JDK 1.1 abgelöst	contains(int, int).
java.awt.Rectangle.inside(int, int)	Bereits im JDK 1.1 abgelöst	contains(int, int).
java.rmi.server.RemoteRef.invoke(RemoteCall)	JDK1.2 Stylestubs verwenden diese Methode nicht länger.	
java.lang.Character.isJavaLetter(char)		isJavaIdentifierStart(char).
java.lang.Character.isJavaLetterOrDigit(char)		isJavaIdentifierPart(char).
java.awt.List.isSelected(int)	Bereits im JDK 1.1 abgelöst	isIndexSelected(int).
java.lang.Character.isSpace(char)		isWhitespace(char).
java.awt.Component.keyDown(Event, int)	Bereits im JDK 1.1 abgelöst	processKeyEvent(KeyEvent).
java.awt.Component.keyUp(Event, int)	Bereits im JDK 1.1 abgelöst	processKeyEvent(KeyEvent).
java.awt.Component.layout()	Bereits im JDK 1.1 abgelöst	doLayout().
java.awt.Container.layout()	Bereits im JDK 1.1 abgelöst	doLayout().
java.awt.ScrollPane.layout()	Bereits im JDK 1.1 abgelöst	doLayout().
java.rmi.server.LoaderHandler.loadClass(String)	Keine direkte Entsprechung.	
java.rmi.server.RMIClassLoader.loadClass(String)		loadClass(String,String).
java.rmi.server.LoaderHandler.loadClass(URL, String)	Keine direkte Entsprechung.	

Methode	Erklärung	Ersetzt durch
java.awt.Component.locate (int, int)	Bereits im JDK 1.1 abgelöst	getComponentAt(int, int).
java.awt.Container.locate (int, int)	Bereits im JDK 1.1 abgelöst	getComponentAt(int, int).
java.awt.Component.location()	Bereits im JDK 1.1 abgelöst	getLocation().
java.rmi.server.LogStream.log (String)	Keine direkte Entsprechung.	
java.awt.Component.lostFocus (Event, Object)	Bereits im JDK 1.1 abgelöst	processFocusEvent (FocusEvent).
java.awt.Component. minimumSize()	Bereits im JDK 1.1 abgelöst	getMinimumSize().
java.awt.Container. minimumSize()	Bereits im JDK 1.1 abgelöst	getMinimumSize().
java.awt.TextField. minimumSize()	Bereits im JDK 1.1 abgelöst	getMinimumSize().
java.awt.TextArea. minimumSize()	Bereits im JDK 1.1 abgelöst	getMinimumSize().
java.awt.List.minimumSize()	Bereits im JDK 1.1 abgelöst	getMinimumSize().
java.awt.TextField. minimumSize(int)	Bereits im JDK 1.1 abgelöst	getMinimumSize(int).
java.awt.List.minimumSize (int)	Bereits im JDK 1.1 abgelöst	getMinimumSize(int).
java.awt.TextArea.minimumSize (int, int)	Bereits im JDK 1.1 abgelöst	getMinimumSize(int, int).
javax.swing.text.View. modelToView(int, Shape)		
java.awt.Component.mouseDown (Event, int, int)	Bereits im JDK 1.1 abgelöst	processMouseEvent (MouseEvent).
java.awt.Component.mouseDrag (Event, int, int)	Bereits im JDK 1.1 abgelöst	processMouseMotionEvent (MouseEvent).
java.awt.Component.mouseEnter (Event, int, int)	Bereits im JDK 1.1 abgelöst	processMouseEvent (MouseEvent).
java.awt.Component.mouseExit (Event, int, int)	Bereits im JDK 1.1 abgelöst	processMouseEvent (MouseEvent).
java.awt.Component.mouseMove (Event, int, int)	Bereits im JDK 1.1 abgelöst	processMouseMotionEvent (MouseEvent).
java.awt.Component.mouseUp (Event, int, int)	Bereits im JDK 1.1 abgelöst	processMouseEvent (MouseEvent).

Methode	Erklärung	Ersetzt durch
java.awt.Component.move(int, int)	Bereits im JDK 1.1 abgelöst	setLocation(int, int).
java.awt.Rectangle.move(int, int)	Bereits im JDK 1.1 abgelöst	setLocation(int, int).
org.omg.CORBA.Principal.name()	Deprecated durch CORBA 2.2.	
org.omg.CORBA.Principal.name(byte[])	Deprecated durch CORBA 2.2.	
java.rmi.server.RemoteRef.newCall(RemoteObject, Operation[], int, long)	JDK1.2 Stylestubs verwenden die Methode nicht länger.	
java.awt.Component.nextFocus()	Bereits im JDK 1.1 abgelöst	transferFocus().
java.awt.datatransfer.DataFlavor.normalizeMimeType(String)		
java.awt.datatransfer.DataFlavor.normalizeMimeTypeParameter(String, String)		
org.omg.CORBA.ServerRequest.op_name()		operation()
org.omg.CORBA.ServerRequest.params(NVList)	Ersatz: Methodenargumente	
java.util.Date.parse(String)	Bereits im JDK 1.1 abgelöst	DateFormat.parse(String s).
java.rmi.server.LogStream.parseLevel(String)	Keine direkte Entsprechung.	
java.awt.Component.postEvent(Event)	Bereits im JDK 1.1 abgelöst	dispatchEvent(AWTEvent).
java.awt.MenuComponent.postEvent(Event)	Bereits im JDK 1.1 abgelöst	dispatchEvent(AWTEvent).
java.awt.Window.postEvent(Event)	Bereits im JDK 1.1 abgelöst	dispatchEvent(AWTEvent).
java.awt.MenuContainer.postEvent(Event)	Bereits im JDK 1.1 abgelöst	dispatchEvent(AWTEvent).
java.awt.Component.preferredSize()	Bereits im JDK 1.1 abgelöst	getPreferredSize().
java.awt.Container.preferredSize()	Bereits im JDK 1.1 abgelöst	getPreferredSize().
java.awt.TextField.preferredSize()	Bereits im JDK 1.1 abgelöst	getPreferredSize().

Methode	Erklärung	Ersetzt durch
java.awt.TextArea. preferredSize()	Bereits im JDK 1.1 abgelöst	getPreferredSize().
java.awt.List.preferredSize()	Bereits im JDK 1.1 abgelöst	getPreferredSize().
java.awt.TextField. preferredSize(int)	Bereits im JDK 1.1 abgelöst	getPreferredSize(int).
java.awt.List.preferredSize (int)	Bereits im JDK 1.1 abgelöst	getPreferredSize(int).
java.awt.TextArea. preferredSize(int, int)	Bereits im JDK 1.1 abgelöst	getPreferredSize(int, int).
org.omg.CORBA.portable. InputStream.read_Principal()	Deprecated durch CORBA 2.2.	
java.io.DataInputStream. readLine()	Diese Methode arbeitet nicht zuverlässig.	BufferedReader.readLine()
java.io.ObjectInputStream. readLine()	Diese Methode arbeitet nicht zuverlässig.	DataInputStream
java.rmi.registry. RegistryHandler.registryImpl (int)	Keine direkte Entsprechung.	
java.rmi.registry. RegistryHandler.registryStub (String, int)	Keine direkte Entsprechung.	
java.rmi.server.RemoteCall. releaseInputStream()	Keine direkte Entsprechung.	
java.rmi.server.RemoteCall. releaseOutputStream()	Keine direkte Entsprechung.	
java.awt.TextArea.replaceText (String, int, int)	Bereits im JDK 1.1 abgelöst	replaceRange(String, int, int).
java.awt.Component.reshape (int, int, int, int)	Bereits im JDK 1.1 abgelöst	setBounds(int, int, int, int).
java.awt.Rectangle.reshape (int, int, int, int)	Bereits im JDK 1.1 abgelöst	setBounds(int, int, int, int).
java.awt.Component.resize (Dimension)	Bereits im JDK 1.1 abgelöst	setSize(Dimension).
java.awt.Component.resize (int, int)	Bereits im JDK 1.1 abgelöst	setSize(int, int).
java.awt.Rectangle.resize (int, int)	Bereits im JDK 1.1 abgelöst	setSize(int, int).
org.omg.CORBA.ServerRequest. result(Any)		set_result

Methode	Erklärung	Ersetzt durch
java.lang.Thread.resume()		
java.lang.ThreadGroup.resume()		
java.lang.Runtime.runFinalizersOnExit(boolean)	Diese Methode ist nicht sicher.	
java.lang.System.runFinalizersOnExit(boolean)	Diese Methode ist nicht sicher.	
java.util.Properties.save(OutputStream, String)	Diese Methode wirft keine IOException aus, auch wenn der Fall eines I/O-Errors bei einem Speichervorgang eintritt. Ersatz seit dem JDK 1.2.	store(OutputStream out, String header)
java.awt.CheckboxGroup.setCurrent(Checkbox)	Bereits im JDK 1.1 abgelöst	setSelectedCheckbox(Checkbox).
java.awt.Frame.setCursor(int)	Bereits im JDK 1.1 abgelöst	Component.setCursor(Cursor).
java.sql.Time.setDate(int)	Bereits im JDK 1.1 abgelöst	Calendar.set(Calendar.DAY_OF_MONTH, int date).
java.util.Date.setDate(int)		
java.rmi.server.LogStream.setDefaultStream(PrintStream)	Keine direkte Entsprechung.	
java.awt.TextField.setEchoCharacter(char)	Bereits im JDK 1.1 abgelöst	setEchoChar(char).
java.sql.Date.setHours(int)	Bereits im JDK 1.1 abgelöst	Calendar.set(Calendar.HOUR_OF_DAY, int hours).
java.util.Date.setHours(int)		
javax.swing.AbstractButton.setLabel(String)		setText(text).
javax.swing.ToolTipManager.setLightWeightPopupEnabled(boolean)	Ursprünglich bis Swing-Version 1.1 verwendet. Das vollständige Swingkonzept wurde überarbeitet.	setToolTipWindowUsePolicy(int).
java.awt.Scrollbar.setLineIncrement(int)	Bereits im JDK 1.1 abgelöst	setUnitIncrement(int).

Methode	Erklärung	Ersetzt durch
java.sql.DriverManager. setLogStream(PrintStream)		
javax.swing.JInternalFrame. setMenuBar(JMenuBar)	Ursprünglich in der Swing-Version 1.0.3 verwendet. Das vollständige Swingkonzept wurde überarbeitet.	setJMenuBar(JMenuBar m).
javax.swing.JRootPane. setMenuBar(JMenuBar)	Ursprünglich in der Swing-Version 1.0.3 verwendet. Das vollständige Swingkonzept wurde überarbeitet.	setJMenuBar(JMenuBar m).
java.sql.Date.setMinutes(int)	Bereits im JDK 1.1 abgelöst	Calendar.set (Calendar.MINUTE, int minutes).
java.util.Date.setMinutes (int)		
java.sql.Time.setMonth(int)	Bereits im JDK 1.1 abgelöst	Calendar.set(Calendar.MONTH, int month).
java.util.Date.setMonth(int)		
java.awt.List. setMultipleSelections (boolean)	Bereits im JDK 1.1 abgelöst	setMultipleMode(boolean).
java.rmi.server.LogStream. setOutputStream(OutputStream)	Keine direkte Entsprechung.	
java.awt.Scrollbar. setPageIncrement(int)	Bereits im JDK 1.1 abgelöst	setBlockIncrement().
java.security.Signature. setParameter(String, Object)		setParameter
java.rmi.server.RemoteStub. setRef(RemoteStub, RemoteRef)	Keine direkte Entsprechung.	
java.sql.Date.setSeconds(int)	Bereits im JDK 1.1 abgelöst	Calendar.set(Calendar.SECOND, int seconds).
java.util.Date.setSeconds (int)		
java.net.MulticastSocket. setTTL(byte)		TimeToLive()
java.net.DatagramSocketImpl. setTTL(byte)		setTimeToLive()

Methode	Erklärung	Ersetzt durch
`java.sql.PreparedStatement.setUnicodeStream(int, InputStream, int)`		
`java.sql.Time.setYear(int)`	Bereits im JDK 1.1 abgelöst	`Calendar.set(Calendar.YEAR, year + 1900).`
`java.util.Date.setYear(int)`		
`java.awt.Component.show()`	Bereits im JDK 1.1 abgelöst	`setVisible(boolean).`
`java.awt.Component.show(boolean)`	Bereits im JDK 1.1 abgelöst	`setVisible(boolean).`
`java.awt.Component.size()`	Bereits im JDK 1.1 abgelöst	`getSize().`
`javax.swing.JTable.sizeColumnsToFit(boolean)`	Ursprünglich in der Swing-Version 1.0.3 verwendet. Das vollständige Swingkonzept wurde überarbeitet.	`sizeColumnsToFit(int).`
`java.lang.Thread.stop()`	Diese Methode gilt mittlerweile nach offizieller Sun-Angabe als unsicher.	
`java.lang.ThreadGroup.stop()`	Diese Methode gilt mittlerweile nach offizieller Sun-Angabe als unsicher.	
`java.lang.Thread.stop(Throwable)`	Diese Methode gilt mittlerweile nach offizieller Sun-Angabe als unsicher.	
`java.lang.Thread.suspend()`	Diese Methode gilt mittlerweile nach offizieller Sun-Angabe als unsicher (Deadlock-Gefahr).	
`java.lang.ThreadGroup.suspend()`	Diese Methode gilt mittlerweile nach offizieller Sun-Angabe als unsicher (Deadlock-Gefahr).	

Methode	Erklärung	Ersetzt durch
`java.util.Date.toGMTString()`	Bereits im JDK 1.1 abgelöst	`DateFormat.format (Date date).`
`java.util.Date. toLocaleString()`	Bereits im JDK 1.1 abgelöst	`DateFormat.format (Date date).`
`java.rmi.server.LogStream. toString()`	Keine direkte Entsprechung.	
`java.rmi.server.Operation. toString()`	Keine direkte Entsprechung.	
`java.io.ByteArrayOutputStream .toString(int)`	Diese Methode arbeitet nicht zuverlässig.	`toString(String enc)`
`java.util.Date.UTC(int, int, int, int, int, int)`	Bereits im JDK 1.1 abgelöst	`Calendar.set(year + 1900, month, date, hrs, min, sec)` oder `GregorianCalendar (year + 1900, month, date, hrs, min, sec).`
`javax.swing.text.View. viewToModel(float, float, Shape)`		
`org.omg.CORBA.portable. OutputStream.write_Principal (Principal)`	Deprecated durch CORBA 2.2.	
`java.rmi.server.LogStream. write(byte[], int, int)`	Keine direkte Entsprechung.	
`java.rmi.server.LogStream. write(int)`	Keine direkte Entsprechung.	

Deprecated-Konstruktoren

Konstruktor	Erklärung	Ersetzt durch
`java.sql.Date(int, int, int)`		`Date(long date).`
`java.util.Date(int, int, int)`	Bereits im JDK 1.1 abgelöst	`Calendar.set(year + 1900, month, date)` oder `GregorianCalendar (year + 1900, month, date).`
`java.util.Date(int, int, int, int, int)`	Bereits im JDK 1.1 abgelöst	`Calendar.set(year + 1900, month, date, hrs, min)` oder `GregorianCalendar (year + 1900, month, date, hrs, min).`
`java.util.Date(int, int, int, int, int, int)`	Bereits im JDK 1.1 abgelöst	`Calendar.set(year + 1900, month, date, hrs, min, sec)` oder `GregorianCalendar (year + 1900, month, date, hrs, min, sec).`
`java.util.Date(String)`	Bereits im JDK 1.1 abgelöst	`DATEFORMAT.PARSE(STRING S).`
`java.rmi.server.Operation (String)`	Keine direkte Entsprechung.	
`java.rmi. RMISecurityException(String)`	Keine direkte Entsprechung.	
`java.rmi. RMISecurityException(String, String)`	Keine direkte Entsprechung.	
`java.rmi. ServerRuntimeException (String, Exception)`	Keine direkte Entsprechung.	
`java.rmi.server. SkeletonMismatchException (String)`	Keine direkte Entsprechung.	
`java.net.Socket(InetAddress, int, boolean)`	Verwendung von `DatagramSocket` anstatt von UDP-Transport.	
`java.net.Socket(String, int, boolean)`	Verwendung von `DatagramSocket` anstatt von UDP-Transport.	

Konstruktor	Erklärung	Ersetzt durch
`java.io.StreamTokenizer (InputStream)`	Bereits im JDK 1.1 soll ein Eingabestrom in einen character-Stream konvertiert werden.	
`java.lang.String(byte[], int)`	Unzuverlässig.	`String` mit Character-encoding.
`java.lang.String(byte[], int, int, int)`	Unzuverlässig.	`String` mit Character-encoding.
`java.sql.Timestamp(int, int, int, int, int, int, int)`		`Timestamp(long millis)`