Christian Ullenboom

Java ist auch eine Insel

Galileo Computing

Liebe Leserin, lieber Leser,

hier ist die neue Java-Insel, mittlerweile schon in der dritten, aktualisierten und erweiterten Auflage.

Christian Ullenboom hat wieder einmal viel Zeit investiert, um die »Insel« noch besser zu machen. Und jede Neuauflage soll ja auch außerdem etwas Neues bieten. Was zeichnet also diese 3. Auflage aus?

Wie immer wurde jedes einzelne Kapitel weiter verbessert, aktualisiert und noch einmal auf Verständlichkeit geprüft. Denn »neu auflegen« soll nicht einfach heißen »ein Kapitel hinzufügen«. Ganz aktuell sind Ausführungen zu Eclipse, **der** Open Source-Plattform führender Software-Hersteller. Eclipse ist wohl auf dem besten Wege, die Java-Welt zu erobern. Dieses interessante Projekt geht an keinem Java-Entwickler mehr vorbei. Und weil sich Fachbücher immer an Ihren Bedürfnissen orientieren sollten, haben wir Eclipse aufgenommen.

Sollten Sie kritische und freundliche Anmerkungen haben, so zögern Sie nicht, sich mit mir in Verbindung zu setzen. Ihre Verbesserungsvorschläge und Ihr Zuspruch sind unentbehrlich für weitere gute Auflagen.

Ich bin gespannt auf Ihre Rückmeldung und wünsche ich Ihnen viel Spaß beim Lesen!

Judith Stevens-Lemoine
Lektorat Galileo Computing
judith.stevens@galileo-press.de
Galileo Press • Gartenstraße 24 • 53229 Bonn

Auf einen Blick

Inhalt

15 Komponenten, Container und Ereignisse 803

Vorwort

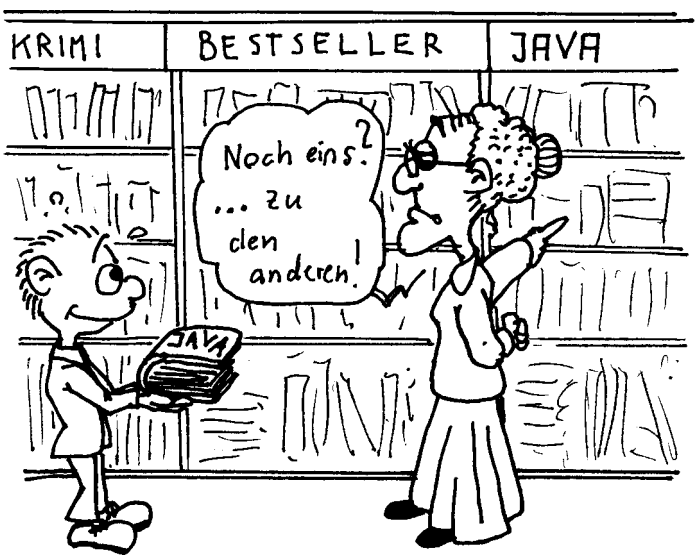
Mancher glaubt schon darum höflich zu sein, weil er sich überhaupt noch der Worte und nicht der Fäuste bedient.
– Hebbel

Java ist auch eine Insel

Java wurde am 23. Mai 1995 auf der SunWorld in San Francisco als neue Programmiersprache vorgestellt. Sie gibt uns elegante Programmiermöglichkeiten; nichts Neues, aber so gut verpackt und verkauft, dass sie angenehm und flüssig zu programmieren ist. Dieses Tutorial beschäftigt sich in 26 Kapiteln mit Java, den Klassen, der Design-Philosophie und der Programmierung.

Motivation für das Buch – oder warum es noch ein Java-Buch gibt ...

Die Beschäftigung mit Java hängt eng mit einer universitären Pflichtveranstaltung zusammen – meiner Projektgruppe zur objektorientierten Dialogspezifikation um das Jahr 1997. Da ich die Teilnehmer davon überzeugen wollte, Java als Programmiersprache einzusetzen (und nicht Objective-C), arbeitete ich meinen ersten Foliensatz für den Seminarvortrag aus. Dieser wurde auch die Basis für meine Schulungsunterlagen, die ich in mehr als fünfzig Kursen immer weiter verbessert habe. Als ich dann noch die Seminararbeit schreiben musste, wurde aus den geplanten Seminarseiten schon ein kleines Buch. Es kam sogar dazu, dass die so genannte Seminararbeit schon sehr viele Seiten fasste und nachher die jetzige Einleitung mehr oder weniger zur Seminararbeit verwurstet wurde; zumal das Tutorial zwischendurch immer dicker geworden ist.



Dass es mich über die universitäre Pflicht hinaus zum Schreiben treibt, ist nur eine Lernstrategie. Wenn ich mich in neue Gebiete einarbeite, lese ich erst einmal quantitativ auf Masse und beginne dann, Zusammenfassungen zu schreiben. Erst beim Schreiben wird mir richtig bewusst, was ich noch nicht weiß. Das Lernen durch Schreiben hat mir auch bei einem anderen Buch sehr geholfen, das leider nicht veröffentlicht wurde. Es ist ein Assembler-Buch für den MC680x0 im Amiga. Doch die Verlage konnten mir nur sagen, dass die Zeit des Amigas vorbei ist. Die Prognosen für Java stehen schon besser, denn der Einsatz von Java in der Wirtschaft fängt gerade erst richtig an. Und ein neues Buch über mein Lieblingsgebiet Performance-Optimierung ist schon in Vorbereitung.

Und für wen ist das Tutorial?

Anfangs war es nicht mein Ziel, ein Buch für eine bestimmte Zielgruppe zu schreiben. Es ist für mich entstanden, um Java gut kennen zu lernen. Nach fünf Jahren Arbeit mit Java und Java-Schulungen (*http://java-tutor.com/seminare/*) haben sich jedoch die Schwerpunkte gewandelt, und das Buch (welches ich immer »Insel« nenne) ist zu einem didaktisch aufbereiteten Lehrbuch geworden, das in seinem Schwierigkeitsgrad nach hinten hin immer komplexer wird. Die unterschiedlichen Kapitel sind für Anfänger der Programmiersprache Java wie auch für Fortgeschrittene konzipiert. Einige Unterkapitel sind für erfahrene Programmierer oder Informatiker. Besonders der Neuling wird an einigen Stellen den sequenziellen Pfad verlassen müssen, da spezielle Kapitel mehr Hintergrundinformationen und Vertrautheit mit Programmiersprachen erfordern. Kenntnisse in einer strukturierten Programmiersprache wie C, Delphi, Visual Basic und ein Verständnis für objektorientierte Technologien sind hilfreich. An einigen Stellen gibt es Verweise auf C(++), diese dienen aber nicht wesentlich dem Verständnis, sondern nur dem Vergleich.

Was das Tutorial nicht ist

Die Java-Technologien und die objektorientierten Techniken sind in den letzten fünf Jahren explodiert. Was zu Anfang noch überschaubar war, ist einer starken Spezialisierung gewichen, so dass es kaum mehr möglich ist, alles in einem Buch zu behandeln. Das möchte die Insel auch auf keinen Fall. Ein Buch, das sich speziell mit der grafischen Oberfläche Swing beschäftigt, ist vom Umfang genauso dick wie die jetzige Insel. Nicht anders verhält es sich mit den anderen Spezialthemen wie Objektorientierte Analyse/Design, UML, Verteilte Programmierung, Datenbankanbindung, Web-Dienste, dynamische Web-Seiten und viele andere Themen. Die Bereiche werden in diesem Buch bis zu einer allgemein verständlichen Tiefe behandelt, und die Wissensneugier muss dann ein Spezialbuch befriedigen.

Ebenfalls darf das Buch auch nicht als Programmierbuch für Einsteiger verstanden werden. Wer noch nie programmiert hat und unter dem Wort Übersetzen in erster Linie einen Dolmetscher versteht, sollte besser ein anderes Buch vorziehen oder parallel lesen.

Welche Software wir nutzen

Als ich 1997 mit Java begann, kamen die ersten Java-Versionen vom Schöpfer Sun auf den Markt. Bis dahin hat sich die Versionsspirale von 1.0 bis aktuell 1.4 gedreht. Als Grundlage dient daher das Java 2 Software Development Kit (kurz SDK) als Referenzimplementierung. Diese lässt sich über *http://www.javasoft.com/products/* beziehen. Das Paket besteht

im Wesentlichen aus einem Compiler, einem Interpreter und einer Online-Hilfe im HTML- oder Win-Help-Format. Das SDK ist für die Plattformen Windows, Linux und Solaris erhältlich. Eine grafische Entwicklungsoberfläche (IDE) ist nicht Teil des Pakets. Ich verlasse mich auch nicht auf einen Hersteller, da der Markt zu dynamisch ist und die Hersteller verschiedene Entwicklergruppen ansprechen. Die Programme lassen sich mit einem einfachen ASCII-Texteditor eingeben und auf der Kommandozeile übersetzen. Diese Form der Entwicklung ist allerdings nicht mehr zeitgemäß, so dass ein grafischer Kommandozeilen-Aufsatz die Programmerstellung vereinfacht. Ich habe mit Eclipse (*http://www.eclipse.org/*) sehr gute Erfahrungen gemacht. Eine weitere Auswahl von Entwicklungswerkzeugen findet der Leser im ersten Kapitel. Für die Entwicklung von Applets ist ein Browser mit Java-Unterstützung wichtig. Zum Testen lässt sich der AppletViewer aus dem JDK verwenden.

Besonders für Java-Buch-Autoren stellt sich die Frage, welches Java 2 SDK und damit welche Bibliotheken beschrieben werden sollen. Ich habe das Problem so gelöst, dass immer die Möglichkeiten des neuesten JDK (zur Zeit 1.4) genutzt werden. Das wirft an einigen Stellen Probleme auf, beispielsweise dann, wenn eine kommerzielle Entwicklungsumgebung genutzt werden soll oder wenn Applets entwickelt werden. Kommerzielle IDEs hinken dem JDK immer etwas hinterher, und die Web-Browser, die Applets darstellen, verstehen nicht immer die aktuellen Versionen. Hier ist der Stand von 1.1 schon gut! Dennoch ist es für mich eine Frage der Zeit, bis sich auch die Neuerungen durchsetzen, und so ist es schwer zu entscheiden, was denn nun alt ist oder nicht. Wenn ich Anmerkungen im Text anheften würde, bliebe die Frage offen, wann ich diese streichen sollte. Galt vor einem Jahr noch 1.1 als Novum, ist dies heute 1.4. Die Leser finden im Buch nur das, was aktuell möglich ist, und nur aus historischen Gründen Verweise auf frühere Lösungen. Für die Didaktik ist die Versionsfrage auch unerheblich, und Softwareentwickler werden die Online-Dokumentationen konsultieren.

Online-Informationen und Aufgaben

Das Buch ist in der aktuellen Version im Internet unter der Adresse

http://www.galileocomputing.de/

erhältlich. Auf der Web-Seite erfährt der Leser Neuigkeiten und Änderungen.

Der Quellcode der Beispielprogramme ist in fast allen Fällen im Buch abgebildet. Ein komprimiertes Zip-Archiv mit allen Beispielen in elektronischer Form ist zusätzlich auf der Web-Seite erhältlich.

Wer eine Programmiersprache lernen möchte, der muss sie wie eine Fremdsprache sprechen. Begleitend gibt es eine Aufgabensammlung auf der CD und mit vielen Musterlösungen auf der Web-Seite

http://java-tutor.com/aufgaben/j/

Die Seite wird in regelmäßigen Abständen aktualisiert, so dass sich dort immer wieder neue Aufgaben und Lösungen vorfinden.

Organisation der Kapitel

Das **Kapitel 1**, *Schon wieder eine neue Sprache?,* zeigt die Besonderheiten der Sprache Java auf. Einige Vergleiche zu anderen populären objektorientierten Sprachen werden gezogen. Die Absätze sind weniger technisch und zeigen auch den geschichtlichen Ablauf. Das Kapitel ist nicht didaktisch aufgebaut, so dass einige Begriffe erst in den weiteren Kapiteln vertieft werden. Ebenso wird hier dargestellt, wie das Java 2 SDK von Sun zu beziehen und zu installieren ist, damit die ersten Programme übersetzt und gestartet werden können.

Technischer wird es in **Kapitel 2**, *Sprachbeschreibung*. Dort werden Variablen, Typen und die imperativen Sprachelemente hervorgehoben. Dabei werden die Grundlagen für jedes Programm mit Anweisungen und Ausdrücken geschaffen. Hier finden auch Fallanweisungen und Schleifen ihren Platz. Das alles geht noch ohne Objektorientierung.

Objektorientiert wird es in **Kapitel 3**, *Klassen und Objekte*. Dabei kümmern wir uns erst einmal um die in der Standardbibliothek vorhandenen Klassen, eigene Klassen werden später entworfen. Die Bibliothek ist so reichhaltig, dass allein mit den vordefinierten Klassen schon viele Programme entworfen werden können. Speziell die bereitgestellten Datenstrukturen lassen sich vielfältig einsetzen.

Wichtig ist für viele Probleme auch *Der Umgang mit Zeichenketten*, der in **Kapitel 4** vorgestellt wird. Die beiden notwendigen Klassen `String` und `StringBuffer` werden eingeführt, und auch ein Abschnitt über reguläre Ausdrücke fehlt nicht. Bei den Zeichenketten müssen Teile ausgeschnitten, erkannt und konvertiert werden. Ein `StringTokenizer` zerlegt Zeichenfolgen aufgrund von Trennern in Teilzeichenketten. Mit Format-Objekten können beliebige Ausgaben in ein gewünschtes Format gebracht werden. Dazu gehört auch die Ausgabe von Dezimalzahlen.

Ein weiteres Feld, dem sich ohne eigene Klasse genähert werden kann, ist *Mathematisches* in **Kapitel 5**. Dazu wird die Klasse `Math` vorgestellt. Sie stellt typische mathematische Funktionen bereit, um etwa trigonometrische Berechnungen durchzuführen. Mit einer weiteren Klasse können Zufallszahlen erzeugt werden. Auch behandelt das Kapitel den Umgang mit beliebig langen Ganz- oder Fließkommazahlen. Für Matrizen wird eine externe Klasse vorgeschlagen.

Mit diesem Vorwissen über Objekterzeugung und Referenzen kann der nächste Schritt gemacht werden. In **Kapitel 6** werden wir dann *Einige Klassen schreiben*. Anhand von Socken und Kleidungsstücken werden Objekteigenschaften modelliert und Benutzt- und Vererbungsbeziehungen aufgezeigt. Wichtige Konzepte wie statische Eigenschaften, Polymorphie, abstrakte Klassen und Schnittstellen (Interfaces) sowie Sichtbarkeit finden dort Platz. Da Klassen in Java auch innerhalb von anderen Klassen liegen können, wird sich ein eigenes Kapitel damit auseinander setzen.

Danach sind die Grundmauern gelegt, und die verbleibenden Kapitel dienen dem Ausbau des bereits erworbenen Wissens. Dies wird in **Kapitel 7**, *Exceptions,* weiter vertieft. Ausnahmen bilden ein wichtiges Rückgrat in Programmen, da sich Fehler nie ausschließen lassen. Da ist es besser, die Behandlung aktiv zu unterstützen und den Programmierer zu zwingen, sich um Fehler zu kümmern und diese zu behandeln. Es wird gezeigt, wie sich eigene Ausnahmen programmieren lassen.

Zu der 2.700 Klassen und Schnittstellen umfassenden Klassenbibliothek gibt **Kapitel 8**, *Die Funktionsbibliothek,* eine Übersicht und führt in wichtige Klassen ein, etwa Konvertieren von Datentypen.

Das **Kapitel 9** beschäftigt sich dann mit *Threads und nebenläufiger Programmierung*. Dabei umfasst das Kapitel auch die Koordination mehrerer kooperierender oder konkurrierender Threads.

Der Rest des Buches vertieft ausgewählte Bereiche. Hier kann der Leser den sequenziellen Pfad verlassen und sich einzelnen Themen widmen. Das beginnt in **Kapitel 10** mit *Raum und Zeit*. Zeitzonen und unterschiedliche Ausgabeformate für Datumswerte werden eingeführt. Darunter fallen auch Datumsberechnungen auf der Grundlage des Gregorianischen Kalenders.

Kapitel 11 beschäftigt sich mit den *Algorithmen und Datenstrukturen*, die die Standardbibliothek anbietet. Die wichtigsten Klassen wie Vektoren, Stapel, Bitmengen und Assoziativspeicher werden vorgestellt und dann unterschiedliche Aufgaben mit den jeweils passenden Datenstrukturen gelöst. Als Algorithmen kommen etwa vorgefertigte Sortierverfahren zum Einsatz.

In **Kapitel 12**, *Datenströme und Dateien,* wird der Fokus auf die Ein-/Ausgabe gelenkt. Zuerst wird gezeigt, wie sich Attribute von Dateien und Verzeichnissen auslesen lassen, und dann, wie sich auf eine Datei wahlfreier Zugriff realisieren lässt. Anschließend folgt der zweite Teil über Datenströme, ein wichtiges Konzept, das auch bei Datenströmen aus Netzwerken, Datenbanken oder Schnittstellen wichtig ist. Die Datenströme können dabei durch Filter geschickt werden. Davon werden einige vorgestellt, die sich zum Beispiel die Zeilennummer merken, einen Datenstrom puffern oder ihn komprimieren. Eine elegante Möglichkeit ist das Serialisieren von Objekten. Dabei wird der Zustand eines Objektes ausgelesen und so in einen Datenstrom geschrieben, dass das Objekt später wiederhergestellt werden kann. Eine eigene Speicherroutine kann so entfallen.

Ein neues Thema spannt **Kapitel 13** mit *XML und Java* auf. Java als plattformunabhängige Programmiersprache und XML als dokumentenunabhängige Beschreibungssprache sind ein ideales Paar, und die Kombination dieser beiden Technologien ist der Renner der letzten Jahre.

Das **Kapitel 14** beschäftigt sich dann mit der *Grafikprogrammierung mit dem AWT.* Das AWT (Abstract Window Toolkit) ist die Java-Möglichkeit, grafische Oberflächen zu gestalten. Dabei gliedert es sich in zwei große Teile: zum einen die direkte Ausgabe von Grafik-Primitiven wie Linien und zum anderen Komponenten für grafische Oberflächen. Das Kapitel behandelt die Themen Fenster, Zeichenketten und Zeichensätze, Farben, Bilder, Bildproduzenten und Konsumenten, Farbmodelle, Bildspeichermöglichkeiten, Filter.

Das anschließende **Kapitel 15** deckt die zweite Aufgabe der grafischen Oberflächen ab: *Komponenten, Ereignisse und Container*. Bei den Komponenten werden Oberflächen-Interaktionskomponenten vorgestellt sowie die Behandlung von Ereignissen, die aus Benutzeraktionen resultieren.

In **Kapitel 16** geht es mit *Netzwerkprogrammierung* weiter. Wir sehen, wie Daten von Web-Servern bezogen werden können und wie eine eigene Client-Server-Kommunikation aufgebaut wird. Bei Web-Servern werden wir CGI-Programme ansprechen, um an gewünschte

Inhalte zu kommen. Neben der gesicherten Verbindung TCP gehen wir auch auf ungesicherte UDP-Verbindungen ein.

Mit dem **Kapitel 17**, *Servlets und Java Server Pages,* geht es dann in die Welt der dynamischen Web-Seiten. Java ist zur Zeit auf der Server-Seite sehr populär und dort besonders beim so genannten Enterprise-Computing. Mit Servlets ist es besonders einfach, dynamische Web-Inhalte zu formulieren, da auf die mitgeschickten Informationen vom Client sehr einfach zugegriffen werden kann. Servlets verfügen zudem über die gesamten Java-Möglichkeiten, insbesondere die Datenbankanbindung.

In **Kapitel 18** zeigen wir auf, wie ein Java-Programm einfach Objekte und Methoden nutzen kann, die auf einem anderen Rechner gespeichert beziehungsweise ausgeführt werden. Solche Programme nutzen die *Verteilte(n) Anwendungen mit RMI*. Dabei wird der Aufruf einer Methode über das Netzwerk übertragen, und für das aufrufende Programm sieht es so aus, als ob es ein normaler Funktionsaufruf für ein lokales Objekt ist.

Das Wissen über die grafischen Oberflächen und Netzwerke wird in **Kapitel 19** über *Applets* verbunden. Das sind Java-Programme, die innerhalb eines Web-Browsers leben. Die Bedeutung von Applets nimmt außerhalb von Firmennetzen (Intranets) immer mehr ab, so dass das Kapitel nicht umfangreich sein muss.

Das Thema *Datenbankmanagement mit JDBC* ist Inhalt von **Kapitel 20**. An einem Beispiel wird gezeigt, wie sich eine Verbindung zu einer Datenbank aufbauen lässt, um dort SQL-Anweisungen abzusetzen, die im nächsten Schritt abgeholt werden. Als Beispieldatenbank wird Interbase von Borland vorgestellt sowie die unbeliebte JDBC-ODBC-Brücke, mit der sich Java-Programme mit jeder ODBC-Datenbank verbinden lassen.

Mit **Kapitel 21** widmen wir uns einer für Java typischen Technik: *Reflection*. Java-Klassen liegen selbst wieder als Meta-Objekte, als Exemplare der speziellen Klasse `Class` vor. Diese `Class`-Objekte geben Auskunft über die verfügbaren und definierten Variablen, Methoden und Konstruktoren. So lassen sich beispielsweise dynamisch bestimmte Methoden aufrufen oder die Werte von dynamisch ausgewählten Objektvariablen abfragen.

Reflection ist eine Vorbereitung auf das **Kapitel 22,** *Komponenten durch Bohnen*. Die wiederverwendbaren Softwarebausteine in Java heißen JavaBeans. Das sind Komponenten, die vor ihrer Benutzung durch ein spezielles Programm visuell editiert und verknüpft werden. Die Beans definieren Eigenschaften (Properties), die über Methodenaufrufe abgefragt und eingestellt werden.

In Java kann nicht alles plattformunabhängig gelöst werden. An einer bestimmen Stelle muss plattformabhängiger Programmcode eingebaut werden. Sun definiert dazu das Java Native Interface (JNI), um auf Nicht-Java-Code wie C(++) zuzugreifen. Wir werden es in **Kapitel 23**, *Java Native Interface,* vorstellen.

Kapitel 24 zeigt kurz *Sicherheitskonzepte*, etwa das Sandkasten-Prinzip, und Sicherheitsmanager auf, aber auch, wie von Daten eine Prüfsumme gebildet werden kann und Daten mit DES verschlüsselt werden.

In **Kapitel 25,** *Dienstprogramme für die Java-Umgebung,* geht es um die zum SDK gehörigen Programme und einige Extratools, die für Ihre Arbeit nützlich sind. Beschrieben werden Compiler, Interpreter und die Handhabung von Jar-Archiven. Dieses Archivformat ist vergleichbar mit den bekannten Zip-Achiven und fasst mehrere Dateien zusammen. Mit den

eingebetteten Dokumentationskommentaren in Java kann aus einer Quellcodedatei ganz einfach eine komplette HTML-Dokumentation der Klassen, Schnittstellen, Vererbungsbeziehungen und Eigenschaften inklusive Verlinkung erstellt werden. Unter den Programmen, die nicht zu einer Standardinstallation gehören, sind etwa Tools, die Java-Programme in C-Programme übersetzen, sie verschönern und Bytecode wieder in lesbaren Java-Quellcode umwandeln.

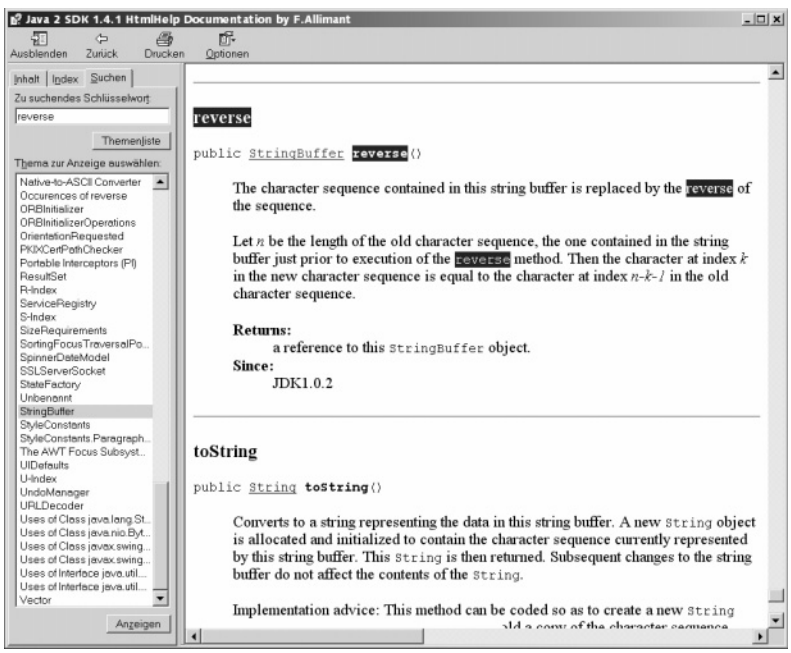
Am Ende des Buches widmet sich **Kapitel 26** einem *Style-Guide*. In Teams sollen sich die Entwickler an Konventionen halten, um das entstehende Softwareprodukt einfacher zu pflegen. Hier werden einige Konventionen vorgestellt, die vom Einrücken von Quellcode bis zur Namensgebung reichen.

Im **Anhang** findet sich das *Quellenverzeichnis* mit Verweisen auf interessante Bücher und Internet-Adressen.

Das Buch in der Lehre einsetzen

Die Insel eignet sich ideal zum Selbststudium. Das erste Kapitel dient zum Warmwerden und plaudert ein wenig über dies und das. Wer auf dem Rechner noch keine Entwicklungsumgebung installiert hat, sollte zuerst das SDK von Sun installieren. Es ist auf der CD oder unter *http://java.sun.com/j2se/* abgelegt.

Zum Entwickeln von Software ist die Hilfe unerlässlich. Da sie einige Megabytes groß ist, muss sie extra ausgepackt werden. Sie ist auf der CD oder unter *http://java.sun.com/docs/* erhältlich. Die API-Dokumentation ist ein einfaches Zip-Archiv mit einer Sammlung von HTML-Dateien, das im Java-Verzeichnis ausgepackt werden kann. Unter *http://java.sun.com/docs/windows_format.html* findet sich die Hilfe auch im HTMLHelp und WinHelp-Format. Das macht das Suchen in der Dokumentation einfacher.



Da das Sun SDK nur Kommandozeilentools installiert, sollte eine grafische IDE (Integrated Development Environment) installiert werden, da dies die Entwicklung von Java-Programmen deutlich komfortabler macht. Eine IDE bietet gegenüber der rohen Kommandozeile einige Vorteile:

- Der Prozess des Editierens, Compilierens, Laufenlassens soll schnell und einfach über einen Tastendruck oder Mausklick angestoßen werden.
- Die Syntax von Java sollte durch Farben hervorgehoben werden (Syntax Highlighting).
- Eine kontextsensitive Hilfe zeigt bei Methoden die Parameter an, und gleichzeitig verweist sie auf die API-Dokumentation.

Weitere Vorteile wie GUI-Builder, Projektmanagement und Debuggen sollen jetzt keine Rolle spielen. Wer neu in die Programmiersprache Java einsteigt, der wird mit CodeGuide seine Freude haben. Es wird im ersten Kapitel ebenfalls beschrieben.

Richtig los geht es in Kapitel 2, und ab dort geht es didaktisch Schritt für Schritt. Wer neu in Java unterwegs ist, wird von Kapitel 2 am meisten profitieren. Wer Kenntnisse in C hat, der kann gleich in Kapitel 3 einsteigen. Wer objektorientiert schon in C++ programmiert hat, kann Kapitel 3 überfliegen und dann einsteigen.

Mit dem Buch und einer Entwicklungsumgebung des Vertrauens lassen sich die ersten Programme entwickeln. Zum Lernen einer neuen Programmiersprache reicht das Lesen aber nicht aus. Zum Trainieren der Fingerfertigkeit dienen die Übungsaufgaben auf der CD. Da Lösungen beigelegt sind, lassen sich die eigenen Lösungen gut mit den Buchübungen vergleichen. Vielleicht bietet die Buchlösung noch eine interessante Lösungsidee oder Alternative an.

Konventionen

In diesem Buch werden folgende Konventionen verwendet: Listings und Methoden sind in `nicht-proportionaler` Schrift gesetzt. Bei Methodennamen folgt immer ein Klammerpaar. Die Parameter sind nicht immer aufgeführt. Neu *eingeführte Begriffe* sind kursiv gesetzt, und der Index verweist genau auf diese Stelle. Des Weiteren sind *Dateinamen*, *Programme* und Dateiendungen (*.txt*) kursiv.

Komplette Programmlistings sind wie folgt aufgebaut:

Listing 0.1 Javaprogrammname.java

```
class Trallala
..
```

Der Quellcode gehört zur Datei *Javaprogrammname.java*. Der Dateiname weicht nur vom Klassennamen ab, wenn sich noch weitere Klassendefinitionen innerhalb einer Datei befinden.

Methoden oder Konstruktoren werden in einer speziellen Auflistung aufgeführt, die ein leichtes Auffinden erlaubt. Im Rechteck steht der voll qualifizierte Klassen- beziehungsweise Schnittstellenname. In den nachfolgenden Zeilen sind geerbte Oberklassen und implementierte Schnittstellen aufgeführt:

```
abstract class java.text.DateFormat
extends Format
implements Cloneable
```

- `Date parse( String ) throws ParseException`
  Parst einen Datum- oder einen Zeit-String.

Da jede Klasse, die keine explizite Oberklasse hat, automatisch von `Object` erbt, ist diese nicht extra angegeben. Die Sichtbarkeit ist, wenn nicht anders angegeben, `public`. Wird eine Schnittstelle beschrieben, sind die Methoden automatisch abstrakt, und das Schlüsselwort `abstract` wird nicht zusätzlich angegeben.

Ausführbare Programme auf der Kommandozeile sind durch ein allgemeines Dollarzeichen am Anfang zu erkennen (auch wenn andere Betriebssysteme und Kommandozeilen ein anderes Prompt anzeigen).

```
$ java ErstesGlueck
```

Danksagungen

Ich hätte gerne einem großen Softwarehaus meinen Dank ausgesprochen, doch leider gibt es keinen Grund dafür. Mit einer Textverarbeitung ist es wie mit Menschen – irgendwie hat doch jeder noch mal eine zweite Chance, auch eine Textverarbeitung. Klappt irgendetwas nicht, nun gut, vielleicht geht es auf einem anderen Weg. Auch meiner Ex-Pommes-Bude nebenan habe ich schon viele Chancen gegeben – aber nichts. Die Pommes blieben weich und pampig. Die Konsequenz ist: Ich gehe nicht mehr hin.

Genauso ist es mit Microsoft Word oder Adobe FrameMaker. Einst war ich von FrameMaker so begeistert, doch das hielt nur einen Monat. Die Texterfassung ist umständlich, und so ging ich zu Word 7 über. Damals waren es schon etwa vierzig Seiten mit Vorlagen. Das Konvertieren ging schnell in drei Tagen über die Bühne. Als ich dann – aus Gründen, die mir heute nicht mehr bekannt sind[1] zu Word 8 überging, ging das Konvertieren schon wieder los. Ich war geblendet von den Funktionen und Spielereien.

Die Ernüchterung kam zwei Monate später. Mein Dokument war auf die Größe von 100 Seiten angeschwollen, und Filialdokumente machten Sinn. Doch plötzlich fehlte eine Datei, andere waren defekt, und Word wollte einfach nicht ohne eine Fehlermeldung die Filialdokumente laden.[2] Sie waren aus unerfindlichen Gründen als fehlerhaft markiert. Auch die Anweisung, alles zu kopieren und in ein neues Dokument zu packen, machte sie nicht wieder einsatzbereit. Dagegen ist das plötzliche Weißwerden des gesamten Texts unter Word 7[3] noch harmlos. Als Word anschließend noch anfing, meine Absatzvorlagen

1 Versionsfanatismus?

2 Aus einem Online-Magazin vom 13. Mai 2002: »A senior Microsoft Corp. executive told a federal court last week that sharing information with competitors could damage national security and even threaten the U. S. war effort in Afghanistan. He later acknowledged that some Microsoft code was so flawed it could not be safely disclosed.«

3 Microsoft hat in den Entwicklungslaboren immense Sicherheitsprobleme. Immer wieder verschwinden Vorversionen von Programmen samt Sicherheitskopien. Dies führt dazu, dass Microsoft unfertige Programmversionen ausliefern und Versionsnummern überspringen musste. Wer über den Aufenthalt von MS Project 2.0, Works 1.0 für Windows, MS DOS 6.1, Excel 1.0, Word 3.0/4.0/5.0 für Windows, Windows NT 1.0/2.0/3.0 Aussagen machen kann, sollte sich bitte an eine Microsoft-Niederlassung wenden.

heiter durcheinander zu bringen und auch nach Ändern und Speichern immer noch die gleichen Effekte zeigte, war es so weit: Word 8 musste weg. Also wieder zurück zu Word 7? Ja! Also RTF, Absatzvorlagen wieder umdefinieren, altes Filialdokument wieder einsetzen. Die Zeit, die ich für Umformatierungen und Konvertierungen verliere, ist weg, und das Einzige, was ich gelernt habe, ist: »Sei vorsichtig bei einem MS-Produkt«! Aber, erzähl' ich damit jemandem etwas Neues?

Nun, ich darf es eigentlich gar nicht erwähnen, aber der Verlag setzt das Buch in FrameMaker. Was sonst?[4] Das Programm eignet sich zwar nicht zur Texterfassung, jedoch ist der Satz sehr gut, die Darstellung von Bild und Text überzeugend schnell und für einen möglichen späteren Druck sehr entgegenkommend. Dort lassen sich auch tausend Seiten mit Bildern und Tabellen ohne Seitenneuberechnung schnell scrollen. So sollte das immer sein. Mich beeindruckt in diesem Zusammenhang immer eine Textverarbeitung auf dem Acorn Archimedes – ich glaube, das war *Impression* von *Computer Concepts*. Sie stellt den Text beim Verschieben eines Bildes mit automatischer Neuberechnung des Textflusses pixelgenau dar. Warum habe ich das in einer PC-Textverarbeitung noch nicht gesehen? Nun denn … Von der Verlagsseite bekomme ich wieder RTF-Dateien aus FrameMaker exportiert, bearbeite sie und gebe sie dann wieder ab. Für die Texterfassung und Korrektur setze ich Word ein – mit roten Kringeln und neuer Rechtschreibung. Ein Glück, dass ich mich über die Umsetzung RTF nach Frame nicht mehr kümmern muss.

Echte Danksagungen

Ich danke besonders Dr. Michael Thies, Uli Holtel, Rolf Leibold, Henryk Plötz, Michael Schulte und Joerg Kulow für ihre Arbeit. Sie haben intensiv gelesen und aktiv mitgewirkt. Ebenso geht ein großes Dankeschön an Hans-Gerd Wiegard, der große Teile des XML-Kapitels weiterentwickelt hat. Die professionellen, aufheiternden Comics stammen von Andreas Schultze (`Akws@aol.com`). Ein weiteres Dankeschön geht an verschiedene treue Leser, deren Namen aufzulisten viel Platz kosten würde. Ich danke auch den vielen Buch- und Artikelautoren für ihre interessanten Werke, aus denen ich mein Wissen über Java aufbauen konnte. Ich danke meinen Eltern für ihre Liebe und Geduld und meinen Freunden/Freundinnen für ihr Vertrauen.

Abschließend möchte ich dem Galileo-Verlag meinen Dank für die Realisierung und unproblematische Zusammenarbeit aussprechen. Für die Zusammenarbeit mit meiner Lektorin Judith bin ich sehr dankbar.

Feedback

Auch wenn wir die Kapitel noch so sorgfältig durchgegangen sind, ist nicht auszuschließen, dass noch Unstimmigkeiten vorhanden sind. Wer Anmerkungen, Hinweise, Korrekturen oder Fragen zu bestimmten Punkten hat, der sollte sich nicht scheuen, mir eine E-Mail unter der Adresse `JavaBuch@Java-Tutor.com` zu senden. Ich bin für Anregung und Kritik stets dankbar.

Und jetzt wünsche ich viel Spaß beim Lesen und Lernen von Java!

Paderborn, im Jahre 2002

Christian U. Ullenboom

4 Ja, o.k., TeX ist auch eine Lösung.

Vorwort Version 2.0

Es sind gerade einmal ein paar Monate her, seitdem die erste Auflage des Buches auf den Markt kam. In der Zeit hatte ich die Möglichkeit, Änderungen für die zweite Version einzubringen. Das sind viele Fehlerkorrekturen besonders optischer Natur. Zudem ist das Buch noch einmal komplett korrigiert worden, was bei der großen Anzahl der blöden Rechtschreibfehler auch sinnvoll war. Der richtig harte Fehler war, dass `Map` die Schnittstelle `Collection` erweitern soll; das ist natürlich Blödsinn. Ich predige das zwar schon seit Jahren in meinen Seminaren, aber wie dieses Missverständnis in das Kapitel kam, bleibt ein Rätsel. Ebenfalls gab es einige Ungenauigkeiten und Fehler bei JSPs. Ich führe das darauf zurück, dass ich sehr altes Info-Material der JSP-Spezifikation 0.9 nutzte. Viele Links mussten ebenfalls aktualisiert werden. Meine Lehre daraus ist: Verweise nie auf Links von Studenten oder Uni-Arbeiten. Die Web-Seiten etablierter Unternehmen sind deutlich stabiler als die einer Privatperson. Eine weitere Änderung betrifft das Kapitel zu Servlets/JSP. Ich habe anfangs die Bedeutung von JSPs völlig falsch eingeschätzt. Das Hauptaugenmerk lag daher in der ersten Auflage auf Servlets, doch das Kapitel ist nun so aufgebaut, dass der Schwerpunkt auf JSPs liegt und Servlets eine untergeordnete Rolle spielen und nur für Binärdaten ihr Dasein fristen. Eine größere Änderung betrifft ebenfalls AWT und Swing. Swing drängt die schwergewichtigen AWT-Komponenten immer mehr in den Hintergrund (obwohl neuere GUIs wie in Eclipse wieder etwas anderes zeigen). Daher habe ich das Kapitel »Komponenten und Container« auf Swing aufgebaut, und die AWT-Komponenten sind raus.

Ein Dankeschön geht wieder an Hans-Gerd, der sich um die Erweiterung des XML-Kapitels gekümmert hat. Götz Wankelmuth nahm sich die Zeit, das Swing-Kapitel zu bearbeiten und mir ausgedruckt zuzuschicken.

Vorwort Version 3.0

Nach dem die Studentenversion der Insel von 1000 Ausgaben innerhalb weniger Wochen verkauft war, lies das erahnen, dass Java noch lange nicht am Ende ist. Nach der Verkauf der Bücher konnte ich daher wieder viele kleine Verbesserungen anwenden und kleine Teile hinzunehmen, unter anderem: Was man in Java nicht machen kann, java und javaw, der Logarithmus, Bits rotieren, BreakIterator, Unterschied zwischen Comparable und Comparator, Wichtige Eigenschaften von Assoziativspeichern, einen HTML-Browser unter Windows aufrufen, Fensterinhalte ändern und die ereignisorientierte Programmierung, Screenshots abspeichern, JComboBox Tastaturbedienung, Swing-Beschriftungen einer anderen Sprache geben, JTable (Spalteninformationen, Tabellenköpfe, Selektionen einer Tabelle), Servlets: Objekte und Dateien per POST verschicken, Datei-Upload.

Wiederum sind hier und da einige Links umgezogen, die aktualisiert wurden. Neue Versionen wurden ebenfalls berücksichtigt. Jürgen Lorenz hat mir einige nette Fehler gemeldet. Danke dafür. UnicastRemoteObjekt heißt natürlich UnicastRemoteObject.

Und jetzt wird es wieder Zeit, dem griechischen Philosophen Platon (427–347 v. Chr.) zu folgen, der sagte: »Der Beginn ist der wichtigste Teil der Arbeit«.

1 Schon wieder eine neue Sprache?

1 Schon wieder eine neue Sprache?

Wir produzieren heute Informationen in Massen, wie früher Autos.
– John Naisbitt

1.1 Der erste Kontakt

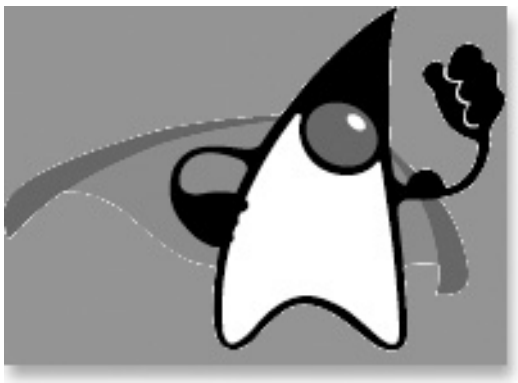
Java ist mittlerweile ein Modewort geworden und ist in aller Munde. Doch nicht so sehr, weil Java[1] eine schöne Insel[2], eine reizvolle Wandfarbe oder eine Pinte mit brasilianischen Rhythmen in Paris ist, sondern vielmehr, weil Java eine neue Programmiersprache ist, mit der »modern« programmiert werden kann. Wer heute nicht mindestens von der »Supersprache« Java gehört hat, scheint megaout.[3] In Java ist viel hineingeredet worden, es wurde als Lösung für alle Softwareprobleme in den Himmel gehoben und als unbrauchbar verdammt und durch zahlreiche »Verbesserungen« verändert.[4] Java ist Philosophie und Innovation gleichzeitig – ein verworrenes Thema. Um in der breiten Softwareentwicklung gegen die vielen konkurrierenden Sprachen die bevorzugte Sprache zu sein, muss Java schon einiges zu bieten haben. Im ersten Kapitel sollen daher kurz die wesentlichen Konzepte der »Internetprogrammiersprache« vorgestellt werden.

1.2 Historischer Hintergrund

In den Siebzigerjahren wollte Bill Joy eine Programmiersprache schaffen, die alle Vorteile von *MESA* und C vereinigt. Diesen Wunsch konnte sich Joy zunächst nicht erfüllen, und erst Anfang der Neunzigerjahre schrieb er in dem Artikel »Further«, wie eine neue objektorientierte Sprache aussehen könnte; sie sollte in den Grundzügen auf C++ aufbauen. Erst später ist ihm bewusst geworden, dass C++ als Basissprache ungeeignet und für große Programme unhandlich ist.

Zu dieser Zeit arbeitete James Gosling an dem *SGML*-Editor »Imagination«. Er entwickelte in C++, und auch er war mit dieser Sprache nicht zufrieden. Aus diesem Unmut entstand die neue Sprache *Oak*. Der Name fiel Gosling ein, als er aus dem Fenster seines Arbeitsplatzes schaute – er sah eine Eiche (engl. oak). Doch vielleicht ist das auch nur eine Legende. Patrick Naughton startete im Dezember 1990 das Green-Projekt, in das Gosling und Mike Sheridan involviert waren. Überbleibsel aus dem Green-Project ist der *Duke*, der zum bekannten Symbol geworden ist.[5]

1 Nicht wieder ein anderes Akronym (Just Another Vague Acronym).
2 Die Insel Java ist die kleinste der Sunda-Inseln in Indonesien mit etwa 90 Millionen Einwohnern. Indonesien selbst hat eine Gesamtbevölkerung von 205 Millionen Einwohnern, die Bevölkerungsdichte in Java ist mit 833 Einwohnern pro Quadratkilometer sehr hoch. Damit verteilt sich auf etwa 7 % der Landoberfläche fast 60 % der Bevölkerung. Hörer der »Drei Fragezeichen« verbinden die Insel vermutlich noch mit Schätzen. Inselinformationen finden sich unter anderem unter *http://www.diht.de/ahk/home/bueros/i/indonesien/standort.html* und *http://www.uemg-omf.ch/laender/indonesien/*.
3 Und die, die in Bewerbungen zehn Jahre Java-Erfahrung angeben, dürfen eh sofort nach Hause gehen.
4 Ein Hinweis an C#
5 Er sieht ein bisschen wie ein Zahn aus und könnte deshalb auch die Werbung eines Zahnarztes sein. Das Design stammt übrigens von Joe Palrang.



Die Idee hinter diesem Projekt war die Entwicklung von Software für interaktives Fernsehen und andere Geräte der Konsumelektronik. Bestandteile dieses Projekts waren das Betriebssystem Green-OS, Goslings Interpreter Oak und einige Hardwarekomponenten. Joy[6] zeigte den Mitgliedern des Green-Projekts seinen Further-Aufsatz und begann mit der Implementierung einer grafischen Benutzeroberfläche. Gosling schrieb den Originalcompiler in C, und anschließend entwarfen Naughton, Gosling und Sheridan den Runtime-Interpreter ebenfalls in C – die Sprache C++ kam nie zum Einsatz. Oak führte die ersten Programme im August 1991 aus. So entwickelte das Green-Dream-Team ein Gerät mit der Bezeichnung *7 (Star Seven), das sie im Herbst 1992 intern vorstellten. Sun-Chef Scott McNealy war von *7 beeindruckt, und aus dem Team wurde im November die Firma First Person, Inc. Nun ging es um die Vermarktung von Star Seven.

Anfang 1993 hörte das Team von einer Anfrage von Time-Warner, die ein System für Set-Top-Boxen brauchten. (Set-Top-Boxen sind elektronische Geräte für Endbenutzer.) First Person richtete den Blick vom Consumer-Markt auf die Set-Top-Boxen. Leider zeigte sich Time-Warner später nicht mehr interessiert, aber First Person entwickelte (sich) weiter. Nach vielen Richtungswechseln konzentrierte sich die Entwicklung auf das World Wide Web (kurz *Web* genannt, selten W^3). Die Programmiersprache sollte Programmcode über das Netzwerk empfangen können, und auch fehlerhafte Programme sollten keinen Schaden anrichten können. Damit konnten die meisten Konzepte aus C(++) schon abgehakt werden – Zugriffe über ungültige Zeiger, die wild den Speicher beschreiben, sind ein Beispiel. Die Mitglieder des ursprünglichen Projektteams erkannten, dass Oak alle Eigenschaften aufwies, die nötig waren, um es im Web einzusetzen – perfekt, obwohl ursprünglich für einen ganz anderen Zweck entwickelt. Die Sprache Oak bekam den Namen »Java«, da der Name »Oak«, wie sich später herausstellte, aus Gründen des Copyrights nicht verwendet werden konnte, da eine andere Programmiersprache schon diesen Namen trug. Nach Überlieferung fiel die Entscheidung für Java in einem Coffeeshop. In Java führte Patrick Naughton den Prototypen des Browsers »WebRunner« vor, der an einem Wochenende entstanden sein soll. Nach etwas Überarbeitung von Jonathan Payne wurde der Browser »HotJava« getauft und im Mai auf der SunWorld '95 der Öffentlichkeit vorgestellt.

6 … oder Gänschen James, Rechnung Freude und Halteseil Steele, wie es Google übersetzt.

Zunächst konnten sich nur wenige Anwender mit HotJava anfreunden. So war es großes Glück, dass Netscape sich entschied, die Java-Technologie zu lizenzieren. Sie wurde in der Version 2.0 des Netscape Navigators implementiert. Der Navigator kam im Dezember 1995 auf den Markt. Im Januar 1996 wurde das JDK 1.0 freigegeben, was den Programmierern die erste Möglichkeit gab, Java-Applikationen und Web-Applets (Applet: »A Mini Application«) zu programmieren. Kurz vor der Fertigstellung des JDK 1.0 gründeten die verbliebenen Mitglieder des Green-Teams die Firma JavaSoft. Und so begann der Siegeslauf.

Version	Datum	Einige Neuerungen/Besonderheiten
JDK 1.0, Urversion	1995	Am 23. Mai 1995 erstmals vorgestellt. Sicherheitsprobleme lösen die weiteren Versionen.
JDK 1.1	Februar 1997	Neuerungen bei der Ereignisbehandlung, Umgang mit Unicode-Dateien, Datenbankunterstützung sowie innere Klassen und eine standardisierte Unterstützung für nicht-Java-Code (nativen Code)
Java 2 SDK 1.2	November 1998	Heißt nun nicht mehr JDK, sondern *Java 2 Software Development Kit* (SDK). Neue Bibliothek für grafische Oberflächen: Swing. Collection
Java 2 SDK 1.3	Mitte 2000	JNDI, RMI/IIOP, Sound-Unterstützung
Java 2 SDK 1.4	Februar 2002	Schnittstelle für XML-Parser, Logging, Neues IO-System (NIO), reguläre Ausdrücke, Assertions
Java 2 SDK 1.5	Ende 2003	Geplant: generische Typen, typsichere Aufzählungen, `foreach`-Konstruktion, Boxing

Es ist lustig zu sehen, dass die Sun-Entwickler für die Java-Versionen immer Tiernamen nutzen, etwa für 1.4.0 Merlin (Falke), 1.4.1 Hopper (Heuschrecke), 1.4.2 Mantis (Gottesanbeterin), 1.5 Tiger. Vielleicht finden wir demnächst auch Java-Flugfrosch, Java-Nashorn und Javaneraffe – diese Tiere gibt es wirklich und einen Java-Tiger auch.

1.3 Eigenschaften von Java

Java ist eine objektorientierte Programmiersprache, die sich durch einige zentrale Eigenschaften auszeichnet. Diese machen sie universell einsetzbar und für die Industrie als robuste Programmiersprache interessant. Da Java objektorientiert ist, spiegelt es den Wunsch der Entwickler wider, moderne und wieder verwertbare Softwarekomponenten zu programmieren.

1.3.1 Bytecode und die virtuelle Maschine

Zunächst ist Java eine Programmiersprache wie jede andere auch. Nur im Gegensatz zu herkömmlichen Übersetzern einer Programmiersprache, die Maschinencode für eine spezielle Plattform generieren, erzeugt der Java-Compiler Programmcode für eine virtuelle Maschine, den so genannten *Bytecode*. Bytecode ist vergleichbar mit Mikroprozessorcode für einen erdachten Prozessor, der Anweisungen wie arithmetische Operationen, Sprünge

und Weiteres kennt. Ein Java-Compiler, etwa der von Sun, der selbst in Java implementiert ist, generiert diesen Bytecode.

Damit aber der Programmcode des virtuellen Prozessors ausgeführt werden kann, führt nach der Übersetzungsphase die *Laufzeitumgebung* (auch *Run-Time-Interpreter* genannt), die *Java Virtuelle Maschine*, den Bytecode aus[7]. Somit ist Java eine compilierte, aber auch interpretierte Programmiersprache – von der Hardwaremethode einmal abgesehen.

Das Interpretieren bereitet noch Geschwindigkeitsprobleme, da das Erkennen, Dekodieren und Ausführen der Befehle Zeit kostet. Im Schnitt sind Java-Programme drei bis zehn Mal langsamer als C(++)-Programme. Die Technik der Just-In-Time(JIT)-Compiler[8] mildert das Problem. Ein JIT-Compiler beschleunigt die Ausführung der Programme, indem die Programmanweisungen der virtuellen Maschine für die physikalische übersetzt werden. Es steht anschließend ein auf die Architektur angepasstes Programm im Speicher, das ohne Interpretation schnell ausgeführt wird. Auch Netscape übernahm im Windows-Communicator[9] 4.0 einen JIT (ein Produkt von ehemals Symantec), um an Geschwindigkeit zuzulegen – obwohl diese Variante noch nicht den gesamten 1.1 Standard beherrschte. (Erst in der Version 4.06 von Netscape kam die volle Unterstützung für Java 1.1.) Mit dieser Technik liegt die Geschwindigkeit zwar in vielen Fällen immer noch unter der von C, aber der Abstand ist geringer.

7 Die Idee des Bytecodes (FrameMaker schlägt hier als Korrekturvorschlag »Bote Gottes« vor) ist schon alt. Die Firma Datapoint schuf um 1970 die Programmiersprache PL/B, die Programme auf Bytecode abbildet. Auch verwendet die Originalimplementierung von UCSD-Pascal, etwa Anfang 1980, einen Zwischencode – kurz p-code.

8 Diese Idee ist auch schon alt: HP hatte um 1970 JIT-Compiler für BASIC-Maschinen.

9 Netscape hört es gar nicht gerne, wenn der Web-Browser als Navigator bezeichnet wird. Hier im Tutorial verwenden wir dies allerdings synonym. Die Firma versteht den Communicator als Web-Lösung, die nicht nur aus einem Web-Browser besteht. Es wird gemunkelt, dass Mitarbeiter aus der Firma rausfliegen, wenn sie das Wort »Navigator« nur in den Mund nehmen ...

Java on a chip

Dieser virtuelle Prozessor wurde mittlerweile auch in Silizium gegossen – eine Entwicklung, die verstärkt von Sun beziehungsweise Lizenznehmern verfolgt wird. Der Prototyp dieses Prozessors (genannt *PicoJava*) ist verfügbar und findet bald Einzug in so genannte Network-Computer. Das sind Computer ohne bewegliche Peripherie, wie Festplatten, die als Terminal am Netz hängen. Bei der Entwicklung des Prozessors stand nicht die maximale Geschwindigkeit im Vordergrund, sondern die Kosten pro Chip, um ihn in jedes Haushaltsgerät einzubauen zu können. Das Interesse an einem Java –on –a Chip ist inzwischen stark zurückgegangen.

Noch mehr Programmiersprachen, die Bytecode erstellen

Doch nicht nur aus der Programmiersprache Java lässt sich Bytecode erzeugen. Zur Zeit gibt es bei verschiedenen Herstellern Entwicklungen von ADA-Compilern und sogar C-Compilern, die Bytecode erstellen. Auch die OOP-Programmiersprache EIFFEL (SmartEiffel)[10] – unter der Leitung von Bertrand Meyer entwickelt – generiert Java-Bytecode. Ebenso gibt es eine Scheme[11]-Umgebung, die komplett in Java programmiert ist. Der Compiler erstellt für den LISP-Dialekt ebenfalls Java-Bytecode. Eine Webseite, die Programmiersprachen für die JVM aufzeigt, ist *http://flp.cs.tu-berlin.de/~tolk/vmlanguages.html*.

Mittlerweile ist Java nicht nur interpretierte Sprache, sondern zugleich auch interpretierende Sprache. Das zeigen unterschiedliche Computer- und Prozessor-Emulationsprogramme.[12]

1.3.2 Kein Präprozessor

In C(++) ersetzt ein Präprozessor Makros etwa für bedingte Compilierung oder Header-Dateien. Einen Präprozessor gibt es in Java nicht und entsprechend keine Header-Dateien. Diese sind in Java nicht nötig, da der Compiler die benötigten Informationen über die Softwareschnittstellen von Klassen direkt aus den Klassendateien liest. Ein schmutziger Trick wie

```
#define private public
#include "allesMoegliche"
```

oder Makros, die Fehler durch doppelte Auswertung erzeugen, sind damit von vornherein ausgeschlossen. Im Übrigen findet sich der Private/Public-Hack im Quellcode von Suns StarOffice. Mit der oberen Definition wird jedes Auftreten von `private` durch `public` ersetzt mit der Konsequenz, dass der Zugriffsschutz ausgehebelt ist.

Leider ist damit auch eine bedingte Compilierung mit `#ifdef` nicht mehr möglich. Dies führt vereinzelt dazu, dass ein externer Präprozessor benutzt werden muss, um den Quellcode entsprechend zu bearbeiten.

10 *http://smarteiffel.loria.fr/*
11 *http://www.adahome.com/Resources/Ada_Java.html*
12 Dies beweist *Hob,* ein portabler ZX-Spectrum-Emulator, der komplett in Java geschrieben ist. Auf der Web-Seite *http://www.emuunlim.com/hob/* gibt es noch viele Spiele dazu, die als Applet ausprobiert werden können. Ebenfalls gibt es den C=64-Emulator-Versuch jc64 unter *http://sourceforge.net/projects/jc64*.

1.3.3 Keine überladenen Operatoren

Wenn wir Operatoren wie das Plus- oder das Minuszeichen verwenden und damit Ausdrücke zusammenfügen, machen wir dies meistens mit bekannten Rechengrößen. So fügt ein Plus zwei Ganzzahlen, aber auch zwei Fließkommazahlen (Gleitkommazahlen) zusammen. Einige Programmiersprachen – meistens Skriptsprachen – erlauben auch das »Rechnen« mit Zeichenketten, mit einem Plus können diese beispielsweise aneinander gehängt werden. Die meisten Programmiersprachen erlauben es jedoch nicht, die Operatoren mit neuer Bedeutung zu versehen und damit Objekte zu verknüpfen. In C++ jedoch ist das Überladen von Operatoren möglich, so dass etwa das Pluszeichen dafür genutzt werden kann, zum Beispiel geometrische Punktobjekte zu addieren. Dies ist praktisch bei umfangreicheren Rechnungen mit Objekten, da dort umständliche Verbindungen nicht über die Methoden geschaffen werden, sondern über ein Operatorzeichen angenehm kurze. Obwohl zuweilen ganz praktisch – das Standardbeispiel sind Objekte für komplexen Zahlen und Brüche –, verführt die Möglichkeit, Operatoren durch den Programmierer zu überladen, oft zu unsinnigem Gebrauch. In Java ist daher das Überladen der Operatoren bisher nicht möglich. Es kann aber sein, dass sich dies in Zukunft ändert.

Die Grundrechenarten sind für Ganzzahlen und Gleitkommazahlen überladen und ebenso ein einfaches Oder, Und oder Xor für Ganzzahlen und boolesche Werte. Der einzige auffällige überladene Operator in Java für Objekte ist das Pluszeichen bei Strings. Zeichenketten können damit leicht zusammengesetzt werden. Informatiker verwenden in dem Zusammenhang auch gerne das Wort Konkatenation (selten Katenation). Bei einem String »Hallo« und »du da« ist »Hallo du da« die Konkatenation der Zeichenketten.

1.3.4 Zeiger und Referenzen

In Java gibt es keine Zeiger (engl. pointer), wie sie aus anderen Programmiersprachen bekannt und gefürchtet sind. Da eine objektorientierte Programmiersprache aber ohne Verweise nicht funktioniert, werden Referenzen eingeführt. Eine Referenz repräsentiert ein Objekt, und eine Variable speichert diese Referenz. Die Referenz hat einen Typ, der sich nicht ändern kann. Ein Auto bleibt ein Auto und kann nicht als Laminiersystem angesprochen werden. Eine Referenz unter Java ist nicht als Zeiger auf Speicherbereiche zu sehen.

Beispiel Dass das Pfuschen in C++ leicht möglich ist und wir Zugriff auf private Elemente über eine Zeigerarithmetik bekommen können, zeigt das folgende Programm. Für uns Java-Programmierer ist dies ein abschreckendes Beispiel.

```
#include <string.h>
#include <iostream.h>

class Ganz_unsicher {
public:
  Ganz_unsicher() { strcpy(passwort, "geheim"); }

private:
  char passwort[100];
};
```

```
void main()
{
  Ganz_unsicher gleich_passierts;
  char *boesewicht = (char*)&gleich_passierts;
  cout << "Passwort: " << boesewicht << endl;
}
```

Diese sehr gepfuschte Art demonstriert, wie problematisch der Einsatz von Zeigern sein kann. Der Zeiger, der zunächst als Referenz auf die Klasse `Ganz_unsicher` gedacht war, mutiert durch die explizite Typumwandlung zu einem Char-Pointer `boesewicht`. Problemlos können über diesen die Zeichen byteweise aus dem Speicher ausgelesen werden. Das erlaubt auch indirekt Zugriff auf die privaten Daten. In Java ist dies nicht möglich, die Implementierung ist sicher, es gibt keinen Zugriff auf private Daten einer Klasse. Zunächst einmal würde der Compiler eine Fehlermeldung geben oder das Laufzeitsystem eine Ausnahme (Exception) auslösen, wenn beispielsweise eine Klasse über das Netz geladen wird.

1.3.5 Bring den Müll raus, Garbage-Collector

In Programmiersprachen wie C++ lässt sich etwa die Hälfte der Fehler auf falsche Speicher-Allokation zurückführen. Arbeiten mit Objekten heißt unweigerlich: Anlegen und Löschen. Die Java-Laufzeitumgebung sorgt sich jedoch selbstständig um die Verwaltung dieser Objekte – die Konsequenz ist: Sie müssen nicht freigegeben werden, ein Garbage-Collector (kurz GC) entfernt sie. Der GC ist Teil des Laufzeitsystems von Java. Das Generieren eines Objekts in einem Block mit anschließender Operation zieht eine Aufräumaktion des GCs nach sich. Nach Verlassen des Wirkungsbereichs erkennt das System das nicht mehr referenzierte Objekt. Ein weiterer Vorteil des GCs: Bei der Benutzung von Unterprogrammen werden oft Objekte zurückgegeben, und in herkömmlichen Programmiersprachen beginnt wieder die Diskussion, welcher Programmteil das Objekt jetzt löschen muss oder ob es nur eine Referenz ist. In Java ist das egal, auch wenn ein Objekt nur Rückgabewert einer Methode ist (anonymes Objekt).

Der GC ist ein spezieller Thread-Prozess, der Objekte markiert, auf die nicht mehr verwiesen wird. Dann entfernt er sie von Zeit zu Zeit. Damit macht der Garbage-Collector die Funktionen `free()` aus C oder `delete()` aus C++ überflüssig. Wir können uns über diese Technik freuen, denn viele Probleme sind damit verschwunden. Nicht freigegebene Speicherbereiche gibt es in jedem größeren Programm, und falsche Destruktoren sind vielfach dafür verantwortlich. An dieser Stelle sollte nicht verschwiegen werden, dass es auch ähnliche Techniken für C(++) gibt.[13]

1.3.6 Ausnahmenbehandlung

Java unterstützt ein modernes System, um mit Laufzeitfehlern umzugehen. In der Programmiersprache wurden Exceptions eingeführt: Objekte, die zur Laufzeit generiert werden und einen Fehler anzeigen. Diese Problemstellen können durch Programmkonstrukte

13 Ein bekannter Garbage-Collector stammt von Hans-J. Boehm, Alan J. Demers und Mark Weiser. Er ist unter *http://reality.sgi.com/boehm_mti/gc.html* zu finden. Der Algorithmus arbeitet jedoch konservativ, das heißt, er findet nicht garantiert alle unerreichbaren Speicherbereiche, sondern nur einige. Eingesetzt wird der Boehm-Demers-Weiser-GC unter anderem in der X11-Bibliothek. Dort sind die `malloc()`- und `free()`-Funktionen einfach durch neue Methoden ausgetauscht.

gekapselt werden. Die Lösung ist in vielen Fällen sauberer als die mit Rückgabewerten und unleserlichen Ausdrücken im Programmfluss. In C++ gibt es ebenso Exceptions, diese werden aber nicht so intensiv wie in Java benutzt.

Aus Geschwindigkeitsgründen wird die Überwachung von Array-Grenzen (engl. Range-Checking) in C(++)[14] nicht durchgeführt. Und der fehlerhafte Zugriff auf das Element `n + 1` eines Felds der Größe n kann zweierlei bewirken: Ein Zugriffsfehler tritt auf, oder, viel schlimmer, andere Daten werden beim Schreibzugriff überschrieben, und der Fehler ist nicht nachvollziehbar. Schon in PASCAL wurde eine Grenzüberwachung mit compiliert. Das Laufzeitsystem von Java überprüft automatisch die Grenzen eines Arrays. Diese Überwachungen können nicht, wie es diverse PASCAL-Compiler erlauben, abgeschaltet werden, sondern sind immer eingebaut. Eine clevere Laufzeitumgebung kann herausfinden, ob keine Überschreitung möglich ist, und diese Abfrage dann wegoptimieren.

1.3.7 Objektorientierung in Java

Die Sprache Java ist nicht bis zur letzten Konsequenz objektorientiert, so wie Smalltalk es vorbildlich zeigt. Primitive Datentypen (beispielsweise Ganzzahlen oder Fließkommazahlen) werden nicht als Objekte verwaltet. Der Grund ist vermutlich in der Performance zu sehen. Der Compiler ist somit besser in der Lage, die Programme zu optimieren.

Java ist als Sprache entworfen worden, die es einfach machen soll, fehlerfreie Software zu schreiben. In C-Programmen erwartet uns statistisch gesehen alle 55 Programmzeilen ein Fehler. Selbst in großen Softwarepaketen, die erst ab einer Million Codezeilen anfangen, findet sich, unabhängig von der zugrunde liegenden Programmiersprache, im Schnitt alle 200 Programmzeilen ein Fehler. Selbstverständlich gilt es, diese Fehler zu beheben, obwohl bis heute noch keine umfassende Strategie für Softwareentwicklung im Großen gefunden wurde. Viele Arbeiten der Informatik beschäftigen sich mit der Frage, wie Tausende Programmierer über Jahrzehnte miteinander arbeiten und Software entwerfen können. Dieses Problem ist nicht einfach zu lösen und wurde im Zuge der Softwarekrise in den Sechzigerjahren heftig diskutiert.

1.3.8 Java-Security-Model

Das Java-Security-Model gewährleistet den sicheren Programmablauf auf verschiedensten Ebenen. Der Verifier liest Code und überprüft die strukturelle Korrektheit und Typsicherheit. Der Klassenlader (engl. class loader) lädt Dateien entweder vom externen Medium wie Festplatte oder auch Netzwerk und überträgt die Java-Binärdaten zum Interpreter. Dort überwacht ein Security-Manager Zugriffe auf das Dateisystem, die Netzwerk-Ports, externe Prozesse und die Systemressourcen. Treten Sicherheitsprobleme auf, so werden diese durch Exceptions zur Laufzeit gemeldet. Das Sicherheitsmodell ist vom Programmierer erweiterbar.

14 In C++ ließe sich eine Variante mit einem überladenen Operator lösen.

1.3.9 Wofür Java nicht geeignet ist

Java-Fanatiker sehen oft nicht, dass Java zwar eine Programmiersprache ist, die für große Anwendungsgebiete geeignet ist, aber auch nicht für alle. Jede Programmiersprache hat ihren Platz – ja, auch Perl.

In erster Linie kann ein Projekt mit Java nicht durchgeführt werden, wenn Eigenschaften gefordert werden, die in Java nicht machbar sind. Java ist plattformunabhängig entworfen worden, so dass alle Funktionen auf allen Systemen lauffähig sind. Benutzerrechte etwa können von Java nicht erfragt oder modifiziert werden, da schon die Rechteverwaltung von Unix und Windows total anders aussehen. Besonders systemnahe Eigenschaften wie Taktfrequenz sind nicht sichtbar und sicherheitsproblematische Manipulationen wie Zugriff auf bestimmte Speicherzellen (das PEEK und POKE) ist ebenso untersagt. Weitere Beschränkungen sind:

- Anzahl freier Bytes im Dateisystem, Laufwerkstyp erkennen (CDRom), CD auswerfen, Verknüpfungen folgen
- Bildschirm auf der Textkonsole löschen, Cursor positionieren und Farben setzen
- Grafische Applikationen mit einem Tray Icon ausstatten
- Auf niedrige Netzwerk-Protokolle wie ICMP zugreifen
- Zugriff auf USB oder Firewire

Für diese Probleme bietet Java keine Bordmittel an. Abhilfe schafft ein nativer Aufruf einer Systemfunktion. Native Funktionen sind Funktionen, die nicht in Java implementiert werden, sondern in einer anderen Programmiersprache, häufig C++. In manchen Fällen lässt sich auch ein externes Programm mit `System.exec()` aufrufen und so etwa die Windows Registry manipulieren und Dateirechte setzen. Es läuft aber immer darauf hinaus, dass für jede Plattform das Problem immer neu implementiert werden muss.

1.4 Java im Vergleich zu anderen Sprachen

Beschäftigen sich Entwickler mit dem Design von Programmiersprachen, so werden häufig existierende Spracheigenschaften auf ihre Tauglichkeit überprüft und dann in das Konzept aufgenommen. Auch Java ist eine fließende und sich entwickelnde Sprache, die viele Merkmale von anderen Sprachen aufweist. Zunächst basierte Java sehr stark auf C++, bis die Entwickler Inkonsistenzen der Sprache nicht übernehmen wollten. Bisweilen wird Java auch als der Nachfolger von C++ gesehen. Auf den ersten Blick erinnert die Syntax sehr stark an C und C++. In der Tat wurden fast alle Anweisungen und Operatoren übernommen. Da viele Konzepte nur von anderen Programmiersprachen übernommen sind, ist die Sprache an sich keine Revolution im Jahr 1996. Java vereinigt vielmehr bekannte und bewährte Konzepte.

Das Klassenkonzept – und damit der objektorientierte (OO-)Ansatz – wurde nicht unwesentlich durch SIMULA und Smalltalk inspiriert. Die Schnittstellen (engl. interfaces), die eine elegante Möglichkeit der Klassenorganisation bieten, sind an Objective-C angelehnt – dort werden sie lediglich *Protocols* genannt. Während in Smalltalk alle Objekte dynamisch verwaltet werden und in C++ der Compiler statisch Klassen zu einem Programm kombiniert, mischt Java auf eine sehr elegante Art und Weise dynamisches und statisches Bin-

den. Klassen können zur Laufzeit geladen werden, Methoden auch auf anderen Rechnern ausgeführt und Ergebnisse über das Netz geschickt werden[15].

Nicht direkt mit der Sprache, aber mit der Anwendung sind die Threads verbunden, das sind leicht zu erzeugende Ausführungsstränge, die unabhängig voneinander arbeiten können. Diese leichtgewichtigen Prozesse gibt es beispielsweise in Solaris.

1.4.1 Java und JavaScript

Obacht ist beim Gebrauch des Namens »Java« geboten. Nicht alles, bei dem Java im Wortstamm auftaucht, hat auch tatsächlich etwas mit Java zu tun; JavaScript hat keinen Bezug zu Java. Die Programmiersprache wurde von Netscape entwickelt. Dazu aus dem Buch »The Java Developer's Resource« ein Zitat: »Java and JavaScript are about as closely related as the Trump Taj Mahal in Atlantic City is to the Taj Mahal in India. In other words Java and JavaScript both have the word Java in their names. JavaScript is a programming language from Netscape which is incorporated in their browsers. It is superficially similar to Java in the same way C is similar to Java but differs in all important respects.«

1.4.2 Normierungsversuche

Ein Normierungsversuch, aus Java eine Programmiersprache unter ISO-Norm zu machen, ist unter anderem daran gescheitert, dass Sun den Namen »Java« weiterverwenden wollte. Dies wäre bei der Norm nicht möglich gewesen. Das Argument ist aber etwas schwach. In der Praxis gibt es auch andere Programme, die bei den Anwendern den offiziellen ISO-Namen tragen und in der Öffentlichkeit ihren ursprünglich aussprechbaren Namen behielten. Die ISO-Norm ist gescheitert. Auch die zweite Normierungsinstanz ECMA lehnte eine Standardisierung ab, da Sun unter allen Umständen verhindern möchte, dass Java in verschiedene Dialekte zerfällt. Aus diesem Grunde hat sich Sun die alleinige Kontrolle über die Weiterentwicklung des Java-Standards vorbehalten. Ein öffentlicher Standard hat aber gerade die Eigenschaft, dass Mitglieder neue Ideen einbringen können. Als Antwort auf Microsofts Forderung, das Java-Warenzeichen allgemein freizugeben, fordert Sun lapidar, dass auch Microsoft die Windows-APIs und das Warenzeichen frei geben sollten. Kein Kommentar von MS!

1.5 Die Rolle von Java im Web

Es ist nicht untertrieben, dem Web eine Schlüsselposition in der Verbreitung von Java zuzuschreiben. Ohne die weltweite Verbreitung des JDK von Sun wäre Java eine Nischenprogrammiersprache mit ungewisser Zukunft. Populär wurde Java in erster Linie durch die Applets, Java-Programme, die vom Browser dargestellt werden. Netscape war eine der ersten Firmen, die einen Java-Interpreter in ihren Web-Browser integrierten. Auch der Internet Explorer (kurz IE) von Microsoft akzeptiert Applets, doch er kommt nicht über die Version 1.1.4 hinaus. Da jedoch der IE sehr verbreitet ist, hat sich die Begeisterung über Applets weitgehend verflüchtigt. Für Softwareentwickler ist es sehr aufwändig, für die gesamte existierende Vielfalt der Java-Versionen zu programmieren.

15 Diese Möglichkeit ist unter dem Namen »RMI« (Remote Method Invocation) bekannt. Bestimmte Methoden können über das Netz miteinander kommunizieren.

Verweise auf Applets werden in eine HTML-Datei eingebettet, und der Browser holt sich eigenständig die Applets über das Netz und führt sie mit seiner virtuellen Maschine aus. Obwohl Applets ganz normale Java-Programme sind, gibt es verständlicherweise einige Einschränkungen. So dürfen Applets nicht – es sei denn, sie sind signiert – auf das Dateisystem zugreifen und wild irgendwelche Dateien löschen, was Java-Applikationen problemlos können.

1.6 Aufkommen von Stand-alone-Applikationen

Obwohl Java durch das Web bekannt geworden ist und dort viele Einsatzgebiete liegen, ist es nicht auf dieses Medium beschränkt. Viele Firmen entdecken ihre Zuneigung zu dieser Sprache und können sich nicht mehr lösen, unter ihnen IBM. Es hat sich gezeigt, dass die Devise »write once, run anywhere« auf der Server-Seite weitgehend zutrifft. Java ist inzwischen wohl die wichtigste Sprache für die Gestaltung von Internet-Applikationen auf dem Server. Sie unterstützt strukturiertes und objektorientiertes Programmieren und ist ideal für größere Projekte, bei denen die Unsicherheiten von C++ vermieden werden sollen.

Nach dem anfänglichen Hype heißt es heute paradoxerweise oft, dass Java zu langsam für Client-Anwendungen sei. Dabei sind die virtuellen Maschinen augrund der Entwicklung von JIT-Compilern und der Hotspot-Technologie in den letzten Jahren sehr viel schneller geworden. Auch die Geschwindigkeit der Prozessoren ist ständig weitergewachsen. Anwendungen wie in Java geschriebene Entwicklungsumgebungen zeigen, dass auch auf der Client-Seite Programme in angemessener Geschwindigkeit laufen können – entsprechend viel Arbeitsspeicher vorausgesetzt. Da ist Java nämlich mindestens so anspruchsvoll wie neue bunte Betriebssystem von MS.

1.7 Entwicklungs- und Laufzeitumgebungen

In der Gründerzeit von Java gab es nur den spartanischen Java-Compiler und die virtuelle Maschine von Sun. Die Situation hat sich geändert, und viele Hersteller stürmen mit Compilern, Laufzeitumgebungen und Integrierten Entwicklungsumgebungen (IDE) auf den Markt.

1.7.1 Aller Anfang mit dem Java SDK

Alles begann mit dem Compiler und der JVM von Sun. Der Compiler unterstützt zwar den gesamten (hausgemachten) Standard, ist aber, da er selbst in Java programmiert ist, langsam. Somit bedarf es für eine Architektur nur eines Interpreters, und schon können mit dem Sun-Compiler Programme übersetzt werden. Sun liefert das Paket aus Compiler und Interpreter kostenlos aus. Sie unterteilen ihr Produkt dabei in drei Bereiche:

- Java 2 Standard Edition (J2SE). Sie ist die normale Softwareumgebung und definiert das Java 2 Software Development Kit, kurz SDK. Geläufig ist immer noch die Bezeichnung JDK (Java Development Kit), die aber mittlerweile veraltet ist.
- Java 2 Enterprise Edition (J2EE). Die Enterprise Edition ist ein Aufsatz auf das J2SE und integriert zusätzliche Pakete wie Enterprise Java Beans, Servlets, JSP, Java-Mail-API, JTS.
- Micro Edition (J2ME). Die Micro Edition ist eine kleine Laufzeitumgebung für kleine PDAs oder Telefone. Für den *PalmPilot* liegt eine Referenzimplementierung vor. Die J2ME löst Personal Java und Embedded Java ab.

Das JDK 1.0.2 wurde 1996 veröffentlicht. Seit den ersten Versionen hat sich die Sprache Java nicht wesentlich verändert. In der Version 1.5 werden wir jedoch große Sprachänderungen erleben.

1.7.2 Die Entwicklungsumgebung von Sun: Sun ONE Studie (früher Forte) und NetBeans

In den Anfängen der Java-Bewegung brachte Sun mit der Software *Java-Workshop* eine eigene Entwicklungsumgebung auf den Markt. Die Produktivitätsmöglichkeiten waren jedoch gering. Das änderte sich, als Sun das kalifornische Softwarehaus Forte übernahm und damit wieder eine bedeutende Rolle bei den Java-Entwicklungsumgebungen einnahm. Sun interessierte sich besonders für Fortes Produkt SynerJ, das im Kern die IDE enthält. So wurde kurze Zeit später *Forte for Java* auf den Markt gebracht. Nun wurde das Produkt wiederum umgetauft und heißt *Sun ONE Studio*.

Sun ONE Studio (*http://wwws.sun.com/software/sundev/jde/index.html*) gibt es in den Ausführungen *Enterprise Edition*, *Mobile Edition* und *Community Edition*. Die Community Edition (sowie die Mobile Edition) stehen zum kostenlosen Download zur Verfügung, die Profiversion kostet 1.995 Dollar. Die Umgebung ist komplett in Java implementiert und modular aufgebaut, so dass Entwickler zusätzliche Bausteine implementieren können. Die Internet und Enterprise Edition unterstützt die J2EE und hilft bei der Erstellung von Datenbank- und JSP/Servlets, EJB, CORBA, RMI und JNDI.

Auf dem gleichen Kern wie Sun ONE Studio baut *NetBeans* auf. Beide verwenden den gleichen Kern, nur ist NetBeans (*http://www.netbeans.org/*) ein Open-Source-Produkt und Sun ONE nicht.

1.7.3 Jikes und Eclipse von IBM

Den Programmierern von IBM, der größten Softwareschmiede der Welt, ist Java mittlerweile so ans Herz gewachsen, dass sie sich von Java nicht trennen können und nur noch damit programmieren wollen. Dabei ist viel Software entstanden, unter anderem der in C++ programmierte Compiler Jikes (*http://www.ibm.com/research/jikes/*), der im Quellcode unter der Open-Source-Lizenz vorliegt. Binärdateien liegen für Win32, Linux, AIX und OS/2 auf dem Server bereit. Für den allgemeinen Einsatz ist Jikes gut geeignet, da er sehr schnell ist, allerdings hat er einige Probleme mit Programmen, die vom Sun-Compiler akzeptiert werden. Dies führt zu spannenden Diskussionen, da IBM von sich behauptet, einen Compiler geschrieben zu haben, der aus Java-Quellcode, definiert in The Java Language Specification (Addison-Wesley 1996) Bytecode erstellt, wie unter The Java Virtual Machine Specification (Addison-Wesley 1996), spezifiziert. Leider ist in einigen Punkten die Sprachspezifikation unzureichend, so dass es zu unterschiedlichen Auslegungen kommt.

Die Entwicklungsumgebung Eclipse

Seit Ende 2001 arbeitet IBM an der neuen unabhängigen Java-basierten Open-Source-Software *Eclipse* (*http://www.eclipse.org/*). IBM löst damit die alte WebSphere-Reihe und die Umgebung *Visual Age for Java* ab. Eclipse macht es möglich, Tools verschiedenster Hersteller zu integrieren. Viele Anbieter haben ihre Produkte schon für Eclipse angepasst, und die

Entwicklung läuft weltweit in einem raschen Tempo. Da Suns IDE NetBeans ebenfalls frei ist und mit anderen Fremdkomponenten bereichert werden kann, zog sich IBM den Groll von Sun zu. Sun wirft IBM vor, die Entwicklergemeinde zu spalten und noch eine unnötige Entwicklungsumgebung auf den Markt zu werfen, wo doch NetBeans schon so toll ist. Nun ja, die Entwickler werden entscheiden. Aktuelle Statistiken zeigen, dass an und mit Eclipse schon mehr Entwickler arbeiten als mit NetBeans.

Eclipse hat gegenüber anderen Umgebungen den Vorteil, dass der Editor besonders Spracheinsteigern hilft, sich mit der Syntax anzufreunden. Dazu unterkringelt Eclipse ähnlich wie moderne Textverarbeitungssysteme fehlerhafte Stellen. Zusätzlich bietet die IDE die notwendigen Hilfen beim Entwickeln, wie etwa automatische Codevervollständigung. Eclipse setzt auf dem Java SDK auf und nutzt somit immer die neuesten Java-Versionen, beziehungsweise beliebige andere Java-Umgebungen.

Eclipse ist ein Java-Produkt mit einer eigenen nativen grafischen Oberfläche, das erstaunlich flüssig seine Arbeit verrichtet – genügend Speicher vorausgesetzt (>256 MB). Die Arbeitszeiten sind auch deswegen so schnell, da Eclipse mit einem so genannten »inkrementellen Compiler« arbeitet. Wenn eine Java-Quellcodedatei gespeichert wird, übersetzt der Compiler automatisch diese Datei; dieses Feature nennt sich »autobuild«.

Bezug und Installation

Eclipse kann unter *http://www.eclipse.org/downloads/index.php* für die Systeme

- Windows 98/ME/2000/XP
- Linux (x86/Motif, x86/GTK 2)
- Mac OSX (Mac/Carbon)
- Solaris 8 (SPARC/Motif)
- QNX (x86/Photon)
- AIX (PPC/Motif)
- HP-UX (HP9000/Motif)

geladen werden. Das ZIP-Archiv wird einfach entpackt und Eclipse aufgerufen. Eine Installation ist nicht erforderlich.

Plugins für Eclipse

Zusätzliche Anwendungen, die in Eclipse integriert werden können, werden *Plugins* genannt. Ein Plugin besteht aus einer Sammlung von Dateien in einem Verzeichnis. Dieses Verzeichnis wird einfach in das Plugin-Verzeichnis von Eclipse kopiert und dann automatisch erkannt und integriert. Bisher sind hunderte von Plugins verfügbar. Die Webseiten *http://www.eclipse.org/community/plugins.html*, *http://www.eclipse-workbench.com/jsp/plugins.jsp* und *http://eclipse-plugins.2y.net/* geben mehr Informationen.

Zu den Nachteilen gehört, dass Eclipse von Haus aus keine Unterstützung von grafischen Oberflächen (GUI-Builder) anbietet. Bei der großen Menge von Plugins ist es jedoch nur eine Frage der Zeit, wann es auch einen freien GUI-Builder gibt.

1.7.4 JBuilder von Borland

Der JBuilder (*http://www.borland.de/jbuilder/*) ist ein Borland-Produkt zur Entwicklung von Java-Applikationen unter Windows, Linux und Solaris. Mit 256 MB Arbeitsspeicher und einem Rechner ab 1 GHz ist die Arbeitsgeschwindigkeit gut. Das Produkt kommt mit Editor, Compiler, Debugger, visuellen Designern und Wizards daher. Die IDE verfügt über zahlreiche Eigenschaften, die den Entwicklungszyklus von professionellen Anwendungen verkürzen. Dazu zählen zum Beispiel ein Debugger, ein GUI-Builder zum Entwurf von grafischen Oberflächen mit voller Swing-Unterstützung, Remote Debugging und viele mehr.

Zur Zeit liegt der JBuilder in der Version 8 und den Produktlinien *Personal*, *SE* (ersetzt die frühere Professional-Version), *Enterprise* und *MobileSet* vor. Alle Versionen nutzen mittlerweile das Java SDK 1.4. Die Personal Version ist eine recht beschnittene, aber dafür frei verfügbare Version (*http://www.borland.com/products/downloads/download_jbuilder.html*). Bei einem Preis von 3.449 Euro für die Enterprise-Version macht sich Borland aber wenig Freunde. Meine persönliche Meinung ist, dass JBuilder durch die immer höher werdenden Kosten die Entwickler vertreibt und Borland sich nicht wundern darf, wenn sich immer mehr Entwickler-Mannschaften gegen den JBuilder entscheiden. Sie werden zur kostengünstigeren Entwicklungsumgebung Sun ONE – die in der Leistungsfähigkeit nicht zurücksteht – wandern oder in Richtung Eclipse plus Plugins gehen.

1.7.5 Together

Ende 2002 hat Borland die Firma Togethersoft[16] (*http://www.togethersoft.com/*) übernommen. Der Hersteller liefert das Case-Tool *Together* (früher TogetherJ) aus, das UML-Design und Codegenerierung komfortabel vereint. Besonders interessant ist das Reverse- und Roundtrip-Engineering, welches es möglich macht, Java-Code zu importieren und UML-Diagramme zu generieren. Die Abbildungen in diesem Buch stammen allesamt von Together. Ein Editor mit Quellcodeerweiterung macht Together zu einer vollständigen IDE. Weiterhin unterstützt das Tool CVS (Concurrent Versioning System), und es lassen sich einfach Datenbankverbindungen über Dialoge aufbauen.

Together spaltet sich in *Control-Center*, *Solo* und *Community Edition* auf. Solo ist die »Einsteigerversion« mit den bekannten UML-Diagrammen Use Case, Sequence, Collaboration, State, Activity, Component und Deployment. Das Control-Center erweitert Solo im Wesentlichen um EJB-Funktionalität. Seit der Version 6 besitzt Together-Control-Center einen GUI-Builder und unterstützt JUnit, EJB2.0 und Web-Services. Die Community-Version (*http://www.togethercommunity.com/tcccommunityedition.jsp*) ist frei, darf aber nicht kommerziell genutzt werden. Da die Software in Java programmiert ist, sollte der Rechner gut mit Speicher ausgestattet sein. Das Programm wird zusammengepackt in einem Archiv geliefert, ist aber auf verschiedenen Plattformen wie Windows und Linux lauffähig.

1.7.6 Die virtuelle Maschine Kaffe von Transvirtual Technologies

Neben der Java VM von Sun liefert die Firma Transvirtual Technologies eine JVM mit dem Namen Kaffe (*http://www.kaffe.org/*) aus, die unter Open Source steht. Der Name wurde

16 Der Objekt-Guru Peter Coad ist Vorsitzender der Firma. Von 1988 bis Ende 1999 lag Together in den Händen des Stuttgarter Softwarehauses Object International. Die Konkurrenz, James Rumbaugh, Grady Booch und Ivar Jacobson (die drei Amigos), sitzen bei Rational Rose.

gewählt, da der Anfang der Entwicklung im Elchland liegt und Kaffe das schwedische Wort für Kaffee ist. Transvirtual Technologies hat zwei JVMs im Angebot: eine offene Version, die Desktop Edition, die unter der GPL steht, und eine Custom Edition für eingebettete Systeme. Insgesamt wird damit eine große Anzahl von Prozessoren (x86, StrongARM, MIPS, m68k, Sparc, Alpha, PowerPC, PARisc) und Betriebssystemen (Embedded Linux, VxWorks, LynxOS, SMX, ThreadX, Linux, DOS, Windows NT 4.0, Windows 98, Windows 2000, Windows CE, Solaris) unterstützt.

Die Implementierung hält sich komplett an den Java-Standard und ist kompatibel zum Personal Java 1.1-System. Sie enthält nicht nur die virtuelle Maschine mit JIT-Compiler, sondern auch native Bibliotheken für Grafik, Ein-/Ausgabe, Dateimanagement und Netzwerkunterstützung. Das Gute dabei ist, dass der Einblick durch Open Source möglich ist. Wir werfen daher immer wieder einen Blick in die Quellcodes und vergleichen Implementierungsstile von Sun und Transvirtual Technologies.

Microsoft hat eine nicht näher benannte Summe in Transvirtual Technologies investiert. Da Microsoft aus Lizenzgründen kein Java mehr entwickeln darf, sichert sich der Konzern den Zugang zu einer moderneren JVM. Denn die Laufzeitumgebung von Kaffe arbeitet nicht nur herkömmliche Java-Programme ab, sondern auch die MS-Erweiterungen, wie J/Direct, die Windows-typische Eigenschaften nutzen. Die Unternehmen Transvirtual, Microsoft und Hewlett-Packard sowie andere kleinere Firmen haben sich zum J-Consortium zusammengeschlossen, um einen alternativen Standard zu etablieren.

Kaffe ist bei einigen Linux-Distributionen schon dabei. Darunter sind Red Hat, Mandrake, SuSE, Debian, FreeBSD, NetBSD, OpenBSD.

1.7.7 Ein Wort zu Microsoft, Java und zu J++

Der Hauptunterschied zwischen dem JDK von Sun und Microsoft liegt darin, dass Applikationen, die unter dem Microsoft Development Kit erstellt wurden, nicht zwangsläufig auf anderen Plattformen wie MacOS, UNIX und Netscape Navigator lauffähig sind. Da Microsoft mal wieder gegen alle Standards ist, sollte der J++-Compiler daher nicht verwendet werden. Microsoft fügte neue Schlüsselwörter (`multicast` und `delegate`) hinzu, entfernte einige Java-Methoden und fügte weitere Methoden und Eigenschaften hinzu. Dies ist zum Beispiel J/Direct. Microsoft versucht, der plattformunabhängigen Programmiersprache den Windows-Stempel zu verpassen. Denn Programme mit J/Direct laufen nur noch unter Windows-Plattformen. Mit J/Direct können Programmierer aus Java heraus direkt auf Funktionen aus dem Win32-API zugreifen und damit reine Windows-Programme in Java programmieren. So haben diese Programme auch Lese- und Schreibzugriff auf die Festplatte – ein Schrecken für alle Java-Programmierer. Durch Integration von DirectX soll die Internet-Programmiersprache Java multimediafähig gemacht werden. Es bleibt abzuwarten, wie es mit der Unterstützung von Java seitens Microsoft weitergeht. Die Aussagen von Microsoft-Projektleiter Ben Slivka über das Java Development Kit beziehungsweise die Java Foundation Classes, man müsse »es bei jeder sich bietenden Gelegenheit anpissen« (»pissing on at every opportunity«), lassen eine harmonische Kooperation mit Sun nicht vermuten.[17]

17 Würden wir nicht gerade im westlichen Kulturkreis leben, wäre diese Geste auch nicht zwangsläufig unappetitlich. Im alten Mesopotamien steht »pissing on« für »anbeten«. Da jedoch die E-Mail nicht aus dem Zweistromland ist, bleibt die wahre Bedeutung wohl unserer Fantasie überlassen.

Suns Produkte, die JNI und RMI nutzen, laufen nicht mit dem IE 4.0 zusammen. Wegen dieser Unregelmäßigkeiten darf Microsoft nach richterlicher Anordnung das Java-Logo nicht mehr in seinen Produkten führen und auch nicht mehr damit werben. Jetzt droht Microsoft natürlich damit, künftige Java-Versionen nur noch bedingt zu unterstützen. Ungefähr zeitgleich nahm Microsoft alle Java-Applets von seinen Web-Seiten – offiziell wegen mangelnder Geschwindigkeit. Da auch der Netscape Navigator Java nicht 100 % unterstützt, wurde das Java-Logo auch aus diesem Internet-Browser entfernt, um einer Klage im Vorfeld aus dem Weg zu gehen. Letztlich hat Sun sich aber durchgesetzt, und Microsofts Java-Variante J++ darf das geschützte Label »100 % Java kompatibel« nicht mehr benutzen.

Ende 2002 verpflichtete der Richter Frederick Motz in Baltimore Microsoft zur Integration einer aktuellen Java-Laufzeitumgebung in Windows. Sun motzte, dass sich Microsoft einen unfairen Vorteil verschafft habe, da in Windows immer nur alte Java-Versionen integriert sind – der Stand Java 1.1.4. Allerdings darf nicht vergessen werden, dass es gerade Sun war, die Microsoft verboten hatten, an Java weiterzuarbeiten. Jetzt bleibt natürlich die Frage, wann eine neue Laufzeitumgebung in Windows integriert wird. Der Richter sah eine Frist von 120 Tagen vor, Java in Windows XP zu integrieren; Startpunkt war der 21. Januar 2003. Die Strafandrohung von 1 Milliarde US-Dollar wurde am 3. Februar zeitweilig außer Kraft gesetzt, da mit einem Berufungsverfahren gerechnet wurde. Wie wir Microsoft kennen, wird die Entscheidung vermutlich so lange hinausgezögert, bis es Java 1.9 gibt.

1.7.8 Direkt ausführbare Programme

Eine in Java geschriebene Applikation lässt sich natürlich nur mit der JVM ausführen. Doch einige Hersteller haben Compiler so ausgelegt, dass sie direkt unter Windows oder einem anderen Betriebssystem ausführbare Programme erstellen. Zwei Modelle werden unterschieden:

- **Native Compiler**. Compiler, die direkt plattformabhängigen Maschinensprachcode erzeugen, heißen *native Compiler*. Unter ihnen sind zum Beispiel *BulletTrain* (*http://www.naturalbridge.com/*), *Excelsior JET* (*http://www.excelsior-usa.com/jet.html*), *JOVE* (*http://www.instantiations.com/jove/product/productdetails.htm*) für Windows und weitere Systeme. Ein freier Compiler unter der GNU-Lizenz ist der GNU-Compiler mit dem Namen *gcj* (*http://gcc.gnu.org/java/*).
- **Wrapper**. Ein Wrapper ist ein ausführbares Programm und liegt wie eine Schale um die Java-Klassen. Der Wrapper ruft dann die virtuelle Maschine auf und übergibt ihr die Klassen. Es ist also immer noch eine Laufzeitumgebung nötig, doch lassen sich den Java-Programmen ein Icon mitgeben und Startparameter setzen.

1.8 Installationsanleitung für das Java 2 SDK unter Microsoft Windows

Damit Java-Programme übersetzt und ausgeführt werden können, müssen wir einen Compiler und Interpreter auf unserem Rechner installiert haben. Das freie JDK von Sun eignet sich zur Entwicklung von einfachen Programmen sehr gut. Die folgende Installationsanleitung beschreibt, wo wir das Java 2 SDK beziehen können und wie es installiert wird.

1.8.1 Das Java 2 SDK beziehen

Es gibt unterschiedliche Möglichkeiten, in den Besitz des Java 2 SDK zu kommen. Wer einen schnellen Zugang zum Internet hat, der kann es sich von den Sun-Seiten herunterladen. Nicht-Internet-Nutzer oder Anwender ohne schnelle Verbindungen finden Entwicklungsversionen sehr häufig auch auf CDs, wie auf unserer in diesem Buch.

Sun bietet auf der Webseite *http://java.sun.com/j2se/1.4/download.html* die Java 2 Platform Standard Edition in der Version 1.4 direkt zum Download für die Versionen Solaris SPARC/x86, Linux x86 und Microsoft Windows an. Die Dokumentation kann ebenfalls von dieser Seite bezogen werden. Bei der Größe des SDK (mehr als 37 MB) musste die Dokumentation getrennt werden, denn auch sie umfasst mehr als 30 MB. Die aktuelle Version des Hauptzweiges ist 1.4.0_03.

1.8.2 Java SDK installieren

Die ausführbare Datei *j2sdk-1_4_x-win.exe* (*x* steht hier als Stellvertreter für die Unterversion) ist das Installationsprogramm. Es installiert die ausführbaren Programme wie Compiler und Interpreter sowie die Bibliotheken, Quellcodes und auch Beispielprogramme.

Während der Installation fragt der Installer nach einem Verzeichnis, er schlägt *C:\j2sdk1.4.0* vor. Nehmen wir im Folgenden den Pfad *C:\Programme\jdk1.4* an. Anschließend folgt ein Dialog, in dem wir Komponenten auswählen können. Wir akzeptieren. Der nachfolgende Dialog möchte WebStart einem Web-Browser wie Microsoft Internet Explorer oder Netscape hinzufügen. Gestatten wir das. Anschließend zeigt sich nach der Komplettinstallation ein Dateibaum mit wenigen Verzeichnissen. Im Unterverzeichnis *bin* befinden sich Compiler und Interpreter. Die Datei *src.jar* enthält den Quellcode der öffentlichen Bibliotheken. Die Datei kann zum Beispiel mit WinZip geöffnet werden, indem *src.jar* in WinZip gezogen wird.

1.8.3 Compiler und Interpreter nutzen

Wir wechseln in die Eingabeaufforderung. Damit sich Programme übersetzen und ausführen lassen, sollten wir den Pfad zum *bin*-Verzeichnis angeben. Im Pfad sind alle Verzeichnisse anzugeben, in der die DOS-Box nach ausführbaren Programmen sucht. Tragen wir den Pfad nicht ein, so müssen wir zum Aufruf immer den kompletten Pfadnamen angeben, was anstrengend ist.

Beispiel Die Klassen im Verzeichnis *D:\Projekte* sollen übersetzt werden.

```
cd D:\projekte
D:\projekte>C:\Programme\jdk1.4\bin\javac *.java
```

Um die Pfade dauerhaft zu setzen, müssen wir die Umgebungsvariable `PATH` modifizieren. Für eine Sitzung reicht es, den *bin*-Pfad hinzuzunehmen.

Beispiel Die aktuelle Pfad-Variable wird modifiziert und zusätzlich auf das bin-Verzeichnis gelegt. Wir setzen *das jdk1.4\bin* am Anfang in den Pfad, damit im Fall von Altinstallationen immer das neue SDK verwendet wird.

```
set PATH=C:\Programme\jdk1.4\bin;%PATH%
```

Damit die Pfadangabe auch nach einem Neustart des Rechners noch verfügbar ist, müssen wir abhängig vom System unterschiedliche Einstellungen vornehmen. Unter Windows NT/2000 aktivieren wir den Dialog **Systemeigenschaften** unter **Start/Einstellungen/Systemsteuerung/System**. Unter dem Reiter **Umgebung** wählen wir bei **Systemvariablen** die Variable `PATH` aus und tragen bei Wert hinter einem Semikolon den Pfad zum *bin*-Verzeichnis ein. Dann können wir den Dialog mit OK verlassen. Falls eine Eingabeaufforderung offen war, wird sie von der Änderung nichts mitbekommen. Ein neue Aufforderung muss geöffnet werden.

Unter Windows 98 editieren wir für den Pfad ebenfalls die Eigenschaft `PATH`, aber diesmal in der Datei *AUTOEXEC.BAT* im Wurzelverzeichnis. In der Regel gibt es schon eine Zeile, die den Pfad setzt. Dort setzen wir das Verzeichnis durch Semikolon getrennt dazu.

Weitere Hilfen gibt die Datei *http://java.sun.com/j2se/1.4/install-windows.html*.

1.9 Das erste Programm compilieren und testen

Nachdem wir die groben Konzepte von Java besprochen haben, wollen wir ganz dem Zitat von Dennis M. Ritchie folgen, der sagt: »Eine neue Programmiersprache lernt man nur, wenn man in ihr Programme schreibt.« Das erste Programm zeigt einen Algorithmus, der die Quadrate der Zahlen von 1 bis 4 ausgibt. Die ganze Programmlogik sitzt in einer Klasse `Quadrat`, die drei Funktionen enthält. Alle Funktionen in einer objektorientierten Programmiersprache wie Java müssen in Klassen platziert werden. Die erste Funktion `quadrat()` bekommt als Übergabeparameter eine ganze Zahl und berechnet daraus die Quadratzahl, die sie anschließend zurückgibt. Eine weitere Funktion übernimmt die Ausgabe der Quadratzahlen bis zu einer vorgegebenen Grenze. Die Funktion bedient sich dabei der Funktion `quadrat()`. Zum Schluss muss es noch ein besonderes Unterprogramm `main()` geben, das für den Java-Interpreter den Einstiegspunkt bietet. Die Methode `main()` ruft dann die Funktion `ausgabe()` auf.

Hinweis Der Java-Compiler unterscheidet sehr penibel zwischen Groß- und Kleinschreibung.

Listing 1.1 Quadrat.java

```
/**
 * @version 1.01    6 Dez 1998
 * @author Christian Ullenboom
 */

public class Quadrat
{
```

```
  static int quadrat( int n )
  {
    return n * n;
  }

  static void ausgabe( int n )
  {
    String s;
    int    i;

    for ( i = 1; i <= n; i=i+1 )
    {
      s = "Quadrat("
          + i
          + ") = "
          + quadrat(i);

      System.out.println( s );
    }
  }

  public static void main( String args[] )
  {
    ausgabe( 4 );
  }
}
```

Ist das Programm unter dem Namen *Quadrat.java* gespeichert, kann es compiliert werden. Die Datei muss so heißen wie ihre Klasse, andernfalls kann das Laufzeitsystem diese nicht finden. Die Beachtung der Groß- und Kleinschreibung ist wichtig. Eine andere Endung wie etwa »*.txt*« oder »*.jav*« ist nicht erlaubt und mündet in einer Fehlermeldung.

```
D:\projekte>javac Quadrat.txt
Quadrat.txt is an invalid option or argument.
Usage: javac <options> <source files>
```

Ist die Datei richtig benannt, so lässt sich der Compiler aufrufen.

```
D:\projekte>javac Quadrat.java
```

Der Compiler legt, vorausgesetzt, das Programm war fehlerfrei, die Datei *Quadrat.class* an. Diese enthält den Bytecode. Mit dem Interpreter kommt das Programm zur Ausführung:

```
D:\projekte>java Quadrat
```

Die erwartete Ausgabe ist

```
Quadrat(1) = 1
Quadrat(2) = 4
Quadrat(3) = 9
Quadrat(4) = 16
```

Als Parameter für den Interpreter wird der Name der Klasse übergeben, die eine `main()`-Funktion enthält und somit als ausführbar gilt. Der Dateiname ist daher auch nicht mit der Endung »class« zu versehen, denn es gibt keine Klasse mit dem Namen *Quadrat.class*. Scheitert die Ausführung des Programms an dem Problem, dass die Klasse nicht verfügbar ist, so muss möglicherweise in die Umgebungsvariable `CLASSPATH` der Punkt (das aktuelle Verzeichnis) mit aufgenommen werden. Ab dem SDK 1.2 ist der `CLASSPATH` allerdings für die Standardausführung nicht nötig und sollte besser gar nicht gesetzt werden.

1.9.1 Häufige Compiler- und Interpreterprobleme

Arbeiten wir auf der Kommandozeilenebene (Shell) ohne eine integrierte Entwicklungsumgebung, können verschiedene Probleme auftreten. Ist der Pfad zum Compiler nicht richtig gesetzt, wird unter DOS (NT-Shell) eine Fehlermeldung der Form

```
D:\>javac Quadrat.java
Der Befehl ist entweder falsch geschrieben oder konnte
 nicht gefunden werden.
Bitte überprüfen Sie die Schreibweise und die
 Umgebungsvariable 'PATH'.
```

ausgegeben. Unter UNIX lautet die Meldung gewohnt kurz:

```
javac: Command not found
```

Findet der Compiler in einer Zeile einen Fehler, so meldet er diesen unter der Angabe der Datei und der Zeilennummer. Nehmen wir noch einmal unser Quadratzahlen-Programm und bauen in der `quadrat()`-Funktion einen Fehler ein – das Semikolon fällt der Löschtaste zum Opfer. Der Compilerdurchlauf meldet:

```
Quadrat.java:10: ';' expected.
    return n * n
               ^
1 error
```

War der Compilerdurchlauf erfolgreich, wird der Interpreter mit dem Programm *java* aufgerufen. Verschreiben wir uns bei dem Namen der Klasse oder fügen wir unserem Klassennamen das Suffix »*class*« zu, so meckert der Interpreter. Beim Versuch, die nicht existente Klasse `Q` zum Leben zu bringen, schreibt der Interpreter auf den Fehlerkanal:

```
java Q.class
Exception in thread "main"  java.lang.NoClassDefFoundError:\
Q/class
```

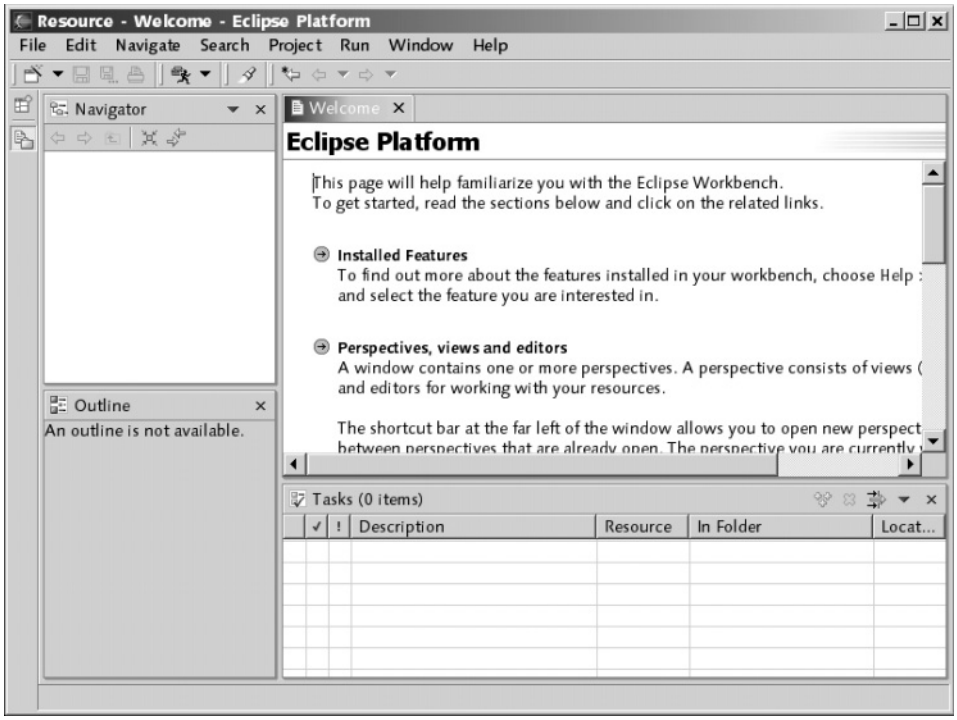
Ist der Name der Klassendatei korrekt, doch die Hauptfunktion hat keine Signatur `public static void main( String [] )`, so kann der Java-Interpreter keine Funktion finden, bei der er mit der Ausführung beginnen soll. Verschreiben wir uns bei der `main()`-Funktion in Quadrat, dann folgt die Fehlermeldung:

```
In class Quadrat: void main(String argv[]) is not defined
```

1.10 Eclipse

1.10.1 Eclipse starten

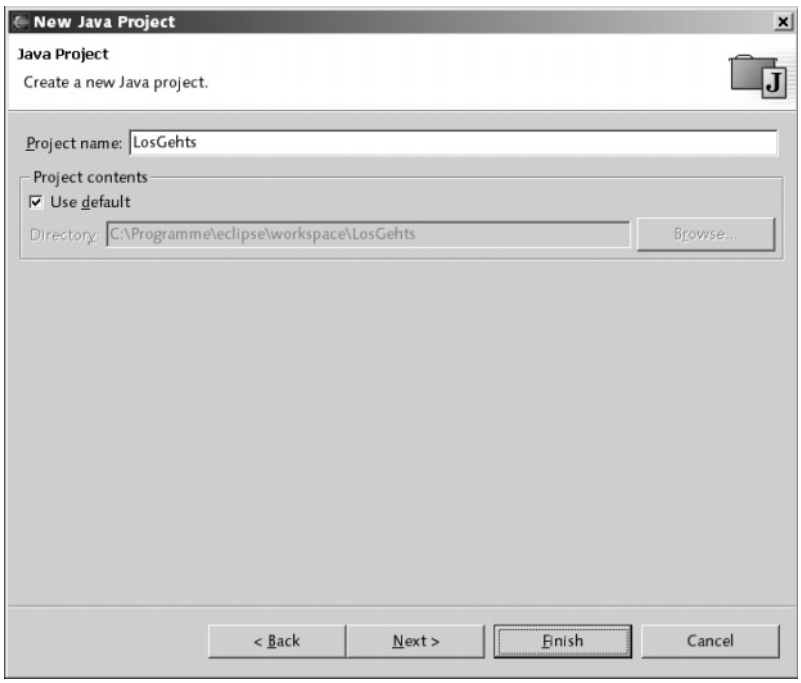
Nach dem Auspacken des Zip-Archivs gibt es im Ordner *eclipse* die ausführbare Datei *eclipse.exe*. Nach dem Start folgt ein Fenster wie dieses:



Das Fenster **Welcome** gibt einige Hilfen, kann aber geschlossen werden.

1.10.2 Das erste Projekt anlegen

Nach dem Start von Eclipse muss ein Projekt angelegt werden – ohne diese kann kein Java-Programm ausgeführt werden. Im Menü ist dazu **File/New/Projekt...** auszuwählen. (Alternativ führt auch die erste Schaltfläche in der Symbolleiste zu diesem Dialog.) Es öffnet sich ein Wizard, der uns ein Java-Projekt erzeugen lässt. Auf der rechten Seite wählen wir dazu **Java Project** und wechseln mit **Next** auf die nächste Einstellung. Wir wollen ein Projekt mit dem Namen **LosGehts** einführen. Mit dem Projekt ist ein Pfad verbunden, in dem die Quellcodes gespeichert sind. Standardmäßig möchte Eclipse die Projekte in einem Unterverzeichnis der Installation anlegen – genau genommen im Unterverzeichnis *workspace*. Das macht nicht in jedem Fall Sinn – über den Schalter **Use defaults** im Rahmen **Project contens** lässt sich ein eigener Pfad entweder von Hand eingeben oder über **Browse** mit einen Dateiauswahldialog ein Verzeichnis auswählen. Die Schaltfläche **Finish** schließt das Anlegen ab.



Nach dem Bestätigen eines kleinen **Confirm Perspective Switch** Dialogs kann es richtig losgehen. Die **Outline** Sicht auf der rechten Seite lässt sich einfach an eine andere Stelle schieben – unter dem **Package Explorer** ist sie meistens gut aufgehoben.

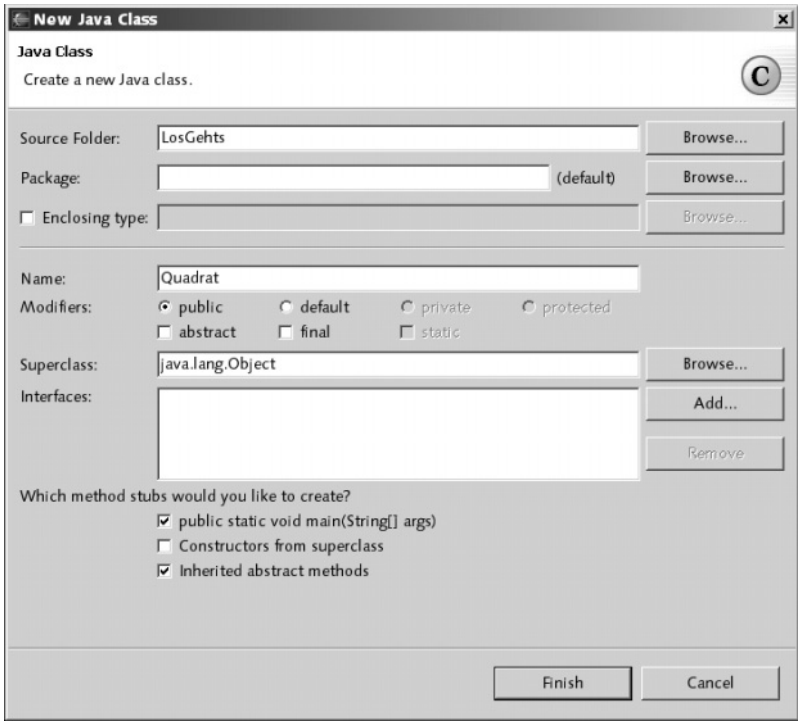
1.10.3 Eine Klasse hinzufügen

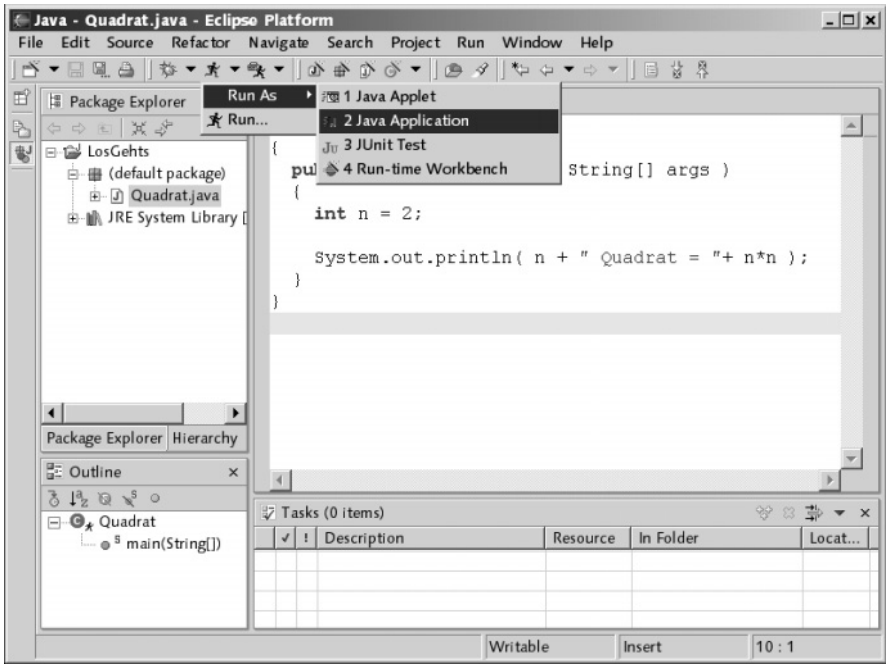
Dem Projekt können nun Dateien, dass heißt Klassen, Grafiken oder andere Inhalte zugeführt werden. Auch können in das Verzeichnis nachträglich Dateien eingeführt werden, die dann direkt von Eclipse anzeigt werden. Doch beginnen wir mit dem Hinzufügen einer Klasse aus Eclipse. Dazu aktiviert der Menüpunkt **File/New/Class** ein neues Fenster (siehe Abbildung Seite 69 oben).

Notwendig ist der Name der Klasse; hier **Quadrat**. Anschließend fügt Eclipse diese Klasse hinzu, erstellt also eine Java-Datei im Dateisystem und öffnet sie gleichzeitig im Editor. Haben wir den Schalter für **public static void main(String[] args)** angewählt, so bekommen wir gleich eine Einstiegsfunktion hinzu, wo sich unser erster Quellcode platzieren lässt.

1.10.4 Übersetzen und Ausführen

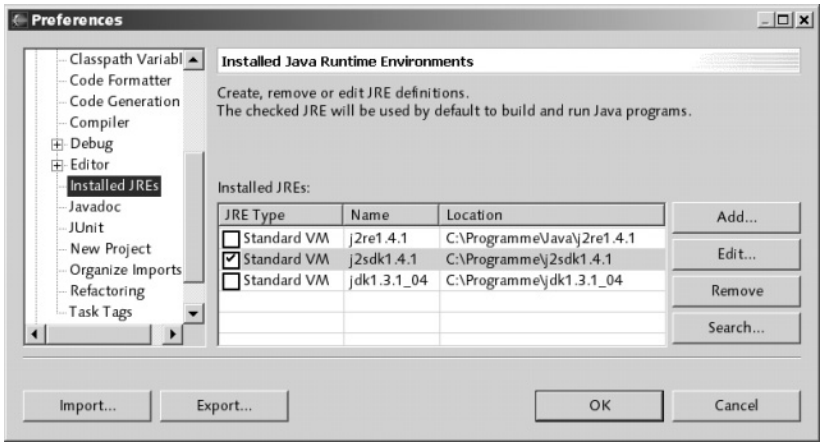
Damit Eclipse eine bestimmte Klasse mit einer **main()**-Funktion ausführt, können wir mehrere Wege gehen. Wird zum ersten Mal eine neue Klasse ausgeführt, können wir auf das Männchen drücken und **Run As** auswählen und anschließend **Java Application** auswählen. Dann startet die Applikation. Assoziierte Eclipse einmal mit einem Start eine Klasse reicht ein Aufruf mit **Strg+F11** (siehe Abbildung Seite 69 unten).

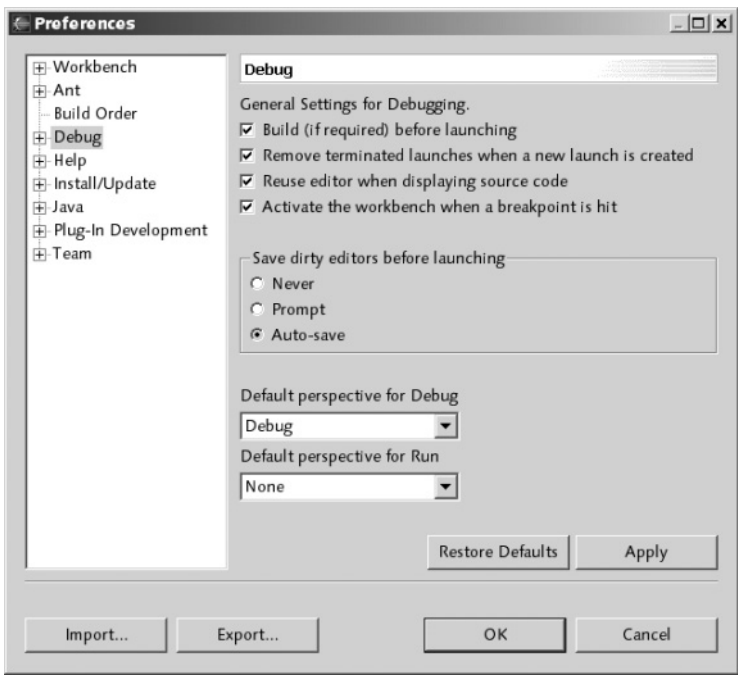

New Java Class
Java Class
Create a new Java class.
Source Folder:
LosGehts
Browse...
Package:
(default)
Browse...
Enclosing type:
Browse...
Name:
Quadrat
Modifiers:
public
default
private
protected
abstract
final
static
Superclass:
java.lang.Object
Browse...
Interfaces:
Add...
Remove
Which method stubs would you like to create?
public static void main(String[] args)
Constructors from superclass
Inherited abstract methods
Finish
Cancel


Java - Quadrat.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Run As
Run...
1 Java Applet
2 Java Application
3 JUnit Test
4 Run-time Workbench
LosGehts
(default package)
Quadrat.java
JRE System Library
String[] args)
int n = 2;
System.out.println(n + " Quadrat = "+ n*n);
Package Explorer
Hierarchy
Outline
Quadrat
main(String[])
Tasks (0 items)
Description
Resource
In Folder
Locat...
Writable
Insert
10 : 1

1.10.5 Nutzen des JDK und Starten ohne Bestätigungsdialog

Beim ersten Start sucht Eclipse eine installierte Java-Version. Das kann eine Version ohne Dokumentation (JRE) oder mit Dokumentation sein (Java SDK). Falls es die JRE ist, fehlt die Dokumentation der Methoden was beim Programmieren sehr unpraktisch ist. Das lässt sich ändern, in dem das SDK eingestellt wird. Unter **Window**, **Preferences** öffnet sich ein globaler Konfigurationsdialog. Im Baum lässt sich **Java** ausfalten und anschließend im Zweig **Installed JREs** nach einem SDK suchen.





Start eines Programmes ohne Speicheraufforderung

In der Standardeinstellung fragt Eclipse nach, geänderte Dateien vor der Übersetzung und Ausführung zu speichern. In der Regel wird bevorzugt, dass eine Entwicklungsumgebung selber die Dateien speichert. Dazu muss eine Einstellung in der Konfiguration vorgenommen werden. Unter **Window**, **Preferences** öffnen wir wieder das Fenster und wählen den Zweig **Debug**. Unter **Save dirty editors before launching** aktivieren wir dann den Schalter **Auto-save**.

2 Sprachbeschreibung

2 Sprachbeschreibung

Wenn ich eine Oper hundertmal dirigiert habe,
dann ist es Zeit, sie wieder zu lernen.
– Arturo Toscanini (1867–1957)

Ein Programm in Java wird nicht umgangssprachlich beschrieben, sondern ein Regelwerk und eine Grammatik definieren die Syntax und die Semantik. In den nächsten Abschnitten werden wir kleinere Beispiele für Java-Programme kennen lernen, und dann ist der Weg frei für größere Programme.

2.1 Anweisungen und Programme

Java zählt zu den imperativen Programmiersprachen, in denen der Programmierer die Abarbeitungsschritte seiner Algorithmen durch *Anweisungen* vorgibt. Diese Befehlsform ist für Programmiersprachen gar nicht selbstverständlich, da es Sprachen gibt, die zu einer Problembeschreibung selbstständig eine Lösung finden. Die Schwierigkeit hierbei liegt darin, die Aufgabe so präzise zu beschreiben, dass das System eine Lösung finden kann. Ein Vertreter dieser Art Sprachen ist Prolog. Auch die Datenbanksprache SQL ist keine imperative Programmiersprache. Denn wie die Datenbank zu unserer Anfrage die Ergebnisse ermittelt, müssen und können wir weder vorgeben noch sehen.

Bleiben wir bei imperativen Sprachen. Um eine Aufgabe zu lösen, müssen wir jeden Abarbeitungsschritt angeben. Abarbeitungsfolgen befinden sich in jedem Kochbuch. Betrachten wir ein ...

Rezept für Flammkuchen

Aus der Hefe, der Milch (lauwarm) und dem Zucker einen dünnflüssigen homogenen Vorteig anrühren. Diesen mit allen anderen Zutaten außer dem Wasser vermischen (am besten mit einem Knethaken oder der Küchenmaschine). So lange Wasser (ca. 2 bis 2 1/2 dl für 500 g Mehl) hinzugeben, bis sich der Teig von der Schüssel (und von den Fingern) löst und sich trotzdem noch feucht anfühlt. Teig zu einem Klumpen formen, mit Mehl bestreuen und mit einem Handtuch abgedeckt ca. 1 Stunde gehen lassen.

Diese Garzeit ist temperaturabhängig. Wenn der Teig in der geheizten Küche auf einen hohen Schrank gestellt wird, reichen manchmal auch nur 30–40 Minuten. Das Teigvolumen sollte sich danach verdoppelt haben. Generell gilt, dass der Teig eher länger als zu kurz gehen sollte.

In der Zwischenzeit wird der Speck gewürfelt, die Zwiebeln werden in feine Ringe geschnitten und der Käse gerieben. Die Crème fraîche wird flüssig gerührt und mit Pfeffer, Muskatnuss und nicht zu viel Salz (da ja schon in Speck, Käse und Teig enthalten) gewürzt. Mit dem Pfeffer und der Muskatnuss nicht zu sparsam sein, die Crème sollte hinterher schon recht würzig sein.

Wenn der Teig gegangen ist, kann man schon mal den Ofen einschalten. Der Teig wird nun in zwei Teile aufgeteilt (einen Teil wieder zurücklegen). Den Teigklumpen nochmals kurz

von Hand durchkneten (vorher die Hände und den Teig leicht bemehlen) und dann auf einer bemehlten Fläche auf Blechgröße ausrollen. Dabei wird der Teig sehr dünn, je dünner umso besser. Den Teig auf das bemehlte Blech legen und mit der Hälfte der Crème bestreichen. Danach den Speck und die Zwiebeln und zum Schluss den Käse darauf geben und den Kuchen im Ofen garen. Nach ca. 10 Minuten, wenn eine leichte Bräunung eintritt, ist der Flammkuchen fertig.

- 500 g Weißmehl
- 40 g Hefe, frisch
- EL Milch
- 1 TL Zucker
- 1 TL Wasser
- 1 TL Salz
- 1 EL Öl
- 200 g Speck, je nach Gusto mehr oder weniger
- Zwiebeln, in feine Ringe geschnitten
- 3 dl Crème fraîche
- 200 g Käse, gerieben
- Pfeffer, frisch gemahlen
- Salz
- Muskatnuss, frisch gerieben

Die Vorschriften kennzeichnen eine klare Sequenz der Tätigkeiten. Dies ist eine wichtige Eigenschaft von Java. Zusätzlich erkennen wir an dieser Arbeitsfolge weitere wichtige Eigenschaften, die sich auf imperative Programmiersprachen übersetzen lassen: Eine Folge von Anweisungen bildet einen Block. Später werden wir diese Operationen zur Vereinfachung des Programms in einen extra Programmblock setzen und »Unterprogramm« nennen.

Des Weiteren lässt sich ein Flammkuchen nicht ohne Kontrolle von Ereignissen zubereiten. Betrachten wir dazu den letzten Satz der Anleitung:

```
Wenn eine leichte Bräunung eintritt, ist der Flammkuchen fertig.
```

Die Abfrage von Gegebenheiten ist sehr wichtig für imperative Sprachen. Genauso Wiederholungen wie

```
So lange Wasser hinzugeben, bis sich der Teig von der Schüssel löst.
```

Hier ist eine Anweisung so lange auszuführen, bis etwas gilt oder nicht mehr gilt. Interessant ist auch folgende Anweisung:

```
Wenn der Teig gegangen ist, kann man schon mal den Ofen einschalten.
```

Dies ist eine Nebenläufigkeit, die in herkömmlichen Sprachen nicht unterstützt wird. Während der Ofen heiß wird, können wir weiterkneten. Java unterstützt ein paralleles Abarbeiten von Programmteilen.

Elementare Anweisungen

Atomare, also unteilbare Anweisungen, werden auch *elementare Anweisungen* genannt. Programme bestehen in der Regel aus mehreren Anweisungen, die dann eine *Anweisungssequenz* ergeben. Die Laufzeitumgebung von Java führt jede einzelne Anweisung der Sequenz in der angegebenen Reihenfolge hintereinander aus.

2.2 Programme

Programme setzen sich aus Anweisungen zusammen. In Java können jedoch nicht einfach Anweisungen in eine Datei geschrieben und dem Compiler übergeben werden. Sie müssen zunächst in einen Rahmen gepackt werden. Dieser Rahmen definiert die Hauptklasse und eine Funktion.

Auch wenn die folgenden Programmcodezeilen am Anfang etwas befremdend wirken, wir werden sie zu einem späteren Zeitpunkt noch genauer erklären. Wir geben der folgenden Datei den Namen *Main.java*.

Listing 2.1 Main.java

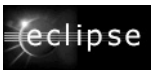
```
class Main
{
  public static void main( String args[] )
  {
  // Hier ist der Anfang unserer Programme

  // Jetzt ist hier Platz für unsere eigenen Anweisungen

  // Hier enden unsere Programme
  }
}
```

Eclipse zeigt Schlüsselwörter, Literale und Kommentare farbig an. Diese Farbgebung lässt sich im Konfigurationsmenü ändern.

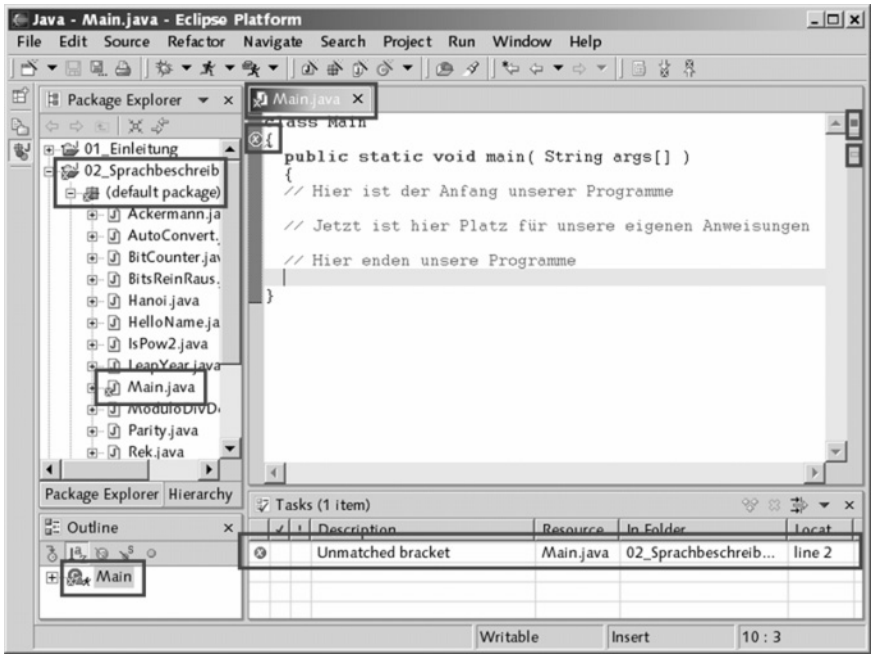
Ein Java-Programm muss immer in einer Klasse definiert sein. Wir haben sie `Main` genannt, der Name ist jedoch beliebig. In geschweiften Klammern folgen benutzerdefinierte Methoden, also Funktionen, die die Klasse anbietet. Eine Funktion ist eine Sammlung von Anweisungen unter einem Namen. Mathematische Funktionen sind mit Funktionen aus Programmiersprachen vergleichbar. Eine Sinus-Funktion schafft es beispielsweise, zu einem gegebenen Wert den Sinus zu berechnen. Wir wissen zwar nicht, wie die Funktion das macht, aber sie kann es. Der Begriff »Funktion« und der objektorientierte Name »Methode« sind in diesem Tutorial synonym verwendet. Vor einer Methode stehen unterschiedliche *Modifizierer*. Der Modifizierer `static` sagt, dass die Methode auch ohne Objekt benutzt werden kann. Wir werden in den folgenden Kapiteln nur mit statischen Methoden arbeiten.

Wir programmieren hier eine besondere Funktion, die sich `main()` nennt. Die Schlüsselwörter davor und die Angabe in dem Paar runder Klammern hinter dem Namen müssen wir einhalten. Die Funktion `main()` ist für die Laufzeitumgebung etwas ganz Besonderes, denn beim Aufruf des Java-Interpreters mit einem Klassennamen wird unsere Funktion als

Erstes ausgeführt.[1] Demnach werden genau die Anweisungen ausgeführt, die innerhalb der geschweiften Klammern stehen. Halten wir uns fälschlicherweise nicht an die Syntax für den Startpunkt, so kann der Interpreter die Ausführung nicht beginnen und wir haben einen semantischen Fehler gemacht, obwohl die Funktion selbst korrekt gebildet ist. Innerhalb von `main()` befindet sich der Name `args`, mit dem die Parameter angesprochen werden. Der Name ist willkürlich, wir werden allerdings immer `args` verwenden.



Eclipse gibt im Falle eines Fehlers sehr viele Hinweise. Ein Fehler im Quellode wird von Eclipse mit einer roten unterkringelten Linie angezeigt. Als weiterer Indikator wird (unter Umständen erst beim Speichern) ein kleines rundes Kreuz an der Fehlerzeile angezeigt. Gleichzeitig findet sich im Schieberegler ein roter kleiner Block. Im **Packager Explorer** findet sich ebenfalls ein Hinweis auf Fehler.



Hinter den beiden Geteilt-Zeichen // befindet sich ein Kommentar. Er gilt bis zum Ende der Zeile und dient dazu, Erläuterungen zu den Quellcodezeilen hinzuzufügen, die ihn verständlicher machen.

Programme übersetzen und starten

Der Quellcode eines Java-Programms ist so allein nicht ausführbar. Ein spezielles Programm, der *Compiler* (auch *Übersetzer* genannt), transformiert das geschriebene Programm in eine andere Repräsentation. Ein Quellcode mit Anweisungen für Programme muss aber nicht zwingend übersetzt werden. Eine andere Gattung für ein Ablaufmodell ist der Inter-

1 Na ja, so ganz präzise ist das auch nicht. In einem `static`-Block könnten wir auch einen Funktionsaufruf setzen, doch das wollen wir hier einmal nicht annehmen. `static`-Blöcke werden beim Laden der Klassen in die virtuelle Maschine ausgeführt. Andere Initialisierungen sind dann auch schon gemacht.

preter. Er liest die Datei Schritt für Schritt ein und führt dann die Anweisungen aus. Der Compiler liest die Datei in einem Rutsch und meldet Fehler. Häufig werden Skriptsprachen interpretiert, früher waren es oft BASIC-Programme. Compiler für geläufige Programmiersprachen wie C(++) oder Delphi übersetzen den Quellcode in Maschinencode. Das ist Binärcode, der vom Prozessor im Computer direkt ausführbar ist. Da unterschiedliche Rechner aber unterschiedliche Prozessoren besitzen, sind die Programme nicht direkt auf verschiedenen Rechnerplattformen ausführbar. Java ist jedoch als Programmiersprache entworfen worden, die plattformunabhängig ist, sich also nicht an einen physikalischen Prozessor klammert. Der Compiler übersetzt den Quellcode nicht in Binärcode für einen konkreten Prozessor, sondern in einen plattformunabhängigen Code, den Bytecode. Prozessoren wie Intel-, AMD- oder G5-CPU können aber mit diesem Bytecode nichts anfangen. Hier hilft ein Interpreter weiter. Dieser liest Anweisung für Anweisung aus dem Bytecode und führt entsprechende Befehle auf dem Mikroprozessor aus. Daher ist Java eine compilierte und interpretierte Sprache zugleich.

Zum Übersetzen der Programme bietet Sun im JDK den Compiler *javac* an. Der Interpreter heißt *java*. Wir haben im einführenden Kapitel über den Ablauf und die möglichen Fehler schon gesprochen.

Beispiel Das Programm mit dem Namen *Main.java* übersetzen und starten:

```
$ javac Main.java
$ java  Main
```

Befindet sich im Quellcode nur die kleinste Ungenauigkeit, die der Compiler nicht toleriert, gibt er einen Fehler aus. Erst wenn Bytecode erzeugt wurde, kann der Interpreter diesen ausführen. Leider sehen wir noch nichts, da wir keine Anweisung gegeben haben. Dies soll sich nun ändern, denn wir wollen lernen, wie eine Bildschirmausgabe aussieht.

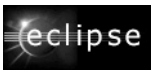
2.2.1 Kommentare

Programmieren heißt nicht nur, einen korrekten Algorithmus in einer Sprache auszudrücken, sondern auch, unsere Gedanken verständlich zu formulieren. Dies geschieht beispielsweise durch eine sinnvolle Namensgebung für Programmobjekte wie Klassen, Funktionen und Variablen. Ein selbsterklärender Klassenname hilft den Entwicklern erheblich. Doch die Lösungsidee und der Algorithmus werden auch durch die schönsten Variablennamen nicht zwingend klarer. Damit Außenstehende (und wir nach Monaten selbst) unsere Lösungsidee schnell nachvollziehen und später das Programm erweitern oder abändern können, werden *Kommentare* in den Quelltext eingeführt. Sie dienen nur den Lesern der Programme, haben aber auf die Abarbeitung keine Auswirkungen.

Kommentarblöcke können durch `/*` und `*/` abgetrennt werden. Zwischen diesen Zeichen kann nahezu jeder beliebige Text stehen. Da es keinen Präprozessor gibt, ist es Aufgabe des Compilers, die Bemerkungen aus dem Quelltext zu entfernen. Da im Einzelfall nur Zeilen auskommentiert werden sollen, ist in Java auch das in C++[2] verwendete `//` erlaubt, welches wir schon kennen gelernt haben.

2 In C++ haben die Entwickler übrigens das Zeilenkommentar // aus der Vor-Vorgängersprache BCPL wieder eingeführt, was in C entfernt wurde.

```
// Zeilenkommentar

/*
 * Blockkommentar
  */
```



Die Tastenkombination Strg+/ kommentiert eine Zeile oder einen Block aus. Eclipse setzt vor die Zeile/Zeilen die Zeichen //. Strg+\ nimmt die Kommentare einer Zeile oder Blöcke wieder zurück.

Dokumentationskommentare

Neben dem Blockkommentar bietet Java eine interessante Möglichkeit der Programmdokumentation. Die Zeichen `/**` und `*/` beschreiben einen so genannten *Dokumentationskommentar* (engl. *Doc-Comment)*, welcher vor Methoden- oder Klassendefinitionen angewendet wird.

Beispiel Dokumentationskommentar:

```
/**
 * Hier drinnen ist ein Doc-Kommentar
 */
```

Der Teil zwischen den Kommentaren wird mit einem externen Dienstprogramm in HTML- oder Framemaker-Dokumente umgewandelt.

Blockkommentare dürfen nicht geschachtelt sein. Ein Zeilenkommentar darf aber im Blockkommentar enthalten sein.

2.2.2 Funktionsaufrufe als Anweisungen

Bisher kennen wir keine konkreten Anweisungen. Daher möchte ich jetzt eine einfache Anweisung vorstellen, die wichtig für Programme ist: der Funktionsaufruf. Allgemein hat er folgendes Format:

```
funktionsname();
```

Innerhalb der Klammern dürfen wir Parameter angeben. Es lassen sich auch Funktionen in dieser Form anwenden, die ein Ausdruck sind und ein Ergebnis zurückgeben, beispielsweise eine Sinus-Funktion. Der Rückgabewert wird dann verworfen. Im Fall der Sinus-Funktion macht das wenig Sinn, denn sie macht außer der Berechnung nichts anderes.

Wir interessieren uns für eine Funktion, die eine Zeichenkette auf dem Bildschirm ausgibt. Sie heißt `println()`. Die meisten Methoden verraten durch ihren Namen, was sie leisten, und für eigene Programme ist es sinnvoll, aussagekräftige Namen zu verwenden. Wenn die Java-Entwickler die Methode `glubschi()`[3] genannt hätten, würde uns der Sinn der Methode verborgen bleiben. `println()` zeigt jedoch durch den Wortstamm »print« an, dass etwas geschrieben wird. Die Endung `ln` (kurz für line) bedeutet, dass noch ein Zeilenvorschubzeichen ausgegeben wird. Umgangssprachlich heißt das: Eine neue Ausgabe beginnt in der

3 Bei uns am Niederrhein steht glubschi für etwas Glitschiges, Schleimiges.

nächsten Zeile. Neben `println()` existiert die Bibliotheksfunktion `print()`, die keinen Zeilenvorschub hervorruft.

Die `printXXX()`[4]-Methoden können in Klammern unterschiedliche Parameter bekommen. Ein Parameter ist ein Wert, den wir der Funktion beim Aufruf mitgeben wollen. Wir wollen die Diskussion über Parameter aber noch etwas verschieben. Unser `printXXX()`-Aufruf soll lediglich *Zeichenketten* ausgeben. (Ein anderes Wort für Zeichenketten ist *String*.) Ein String ist eine Folge von Buchstaben, Ziffern und Sonderzeichen in doppelten Anführungszeichen:

```
"Ich bin ein String"
"Ich auch. Und ich koste 7.59 _"
```

Um die obere Ausgabe mit dem Funktionsaufruf und einem trennenden Zeilenvorschub auf den Bildschirm zu bekommen, schreiben wir:

```
System.out.println( "Ich bin ein String" );
System.out.println();               // Gibt eine Leerzeile aus
System.out.println( "Ich auch. Und ich koste 7.59 _" );
```

Auch wenn eine Funktion keine Parameter erwartet, muss beim Aufruf hinter dem Funktionsnamen ein Klammerpaar folgen. Dies ist konsequent, da wir so wissen, dass es ein Funktionsaufruf ist und nichts anderes. Andernfalls führt es zu Verwechslungen mit Variablen.

Eine weitere Eigenschaft von Java wird ebenfalls an dem Funktionsaufruf sichtbar. Es gibt Methoden, die unterschiedliche Parameter (eine andere Anzahl oder einen unterschiedlichen Typ) besitzen, aber gleichen Namen tragen. Diese Funktionen nennen wir *überladen*. Die `printXXX()`-Methoden sind sogar vielfach überladen und akzeptieren neben Strings auch weitere Werte als Parameter.

Programmieren wir eine ganze Java-Klasse, die etwas auf dem Bildschirm ausgibt.

Listing 2.2 Main.java

```
class Main
{
  public static void main( String args[] )
  {
  // Hier ist der Anfang unserer Programme

    System.out.println( "Hallo Javanesen" );

  // Hier enden unsere Programme
  }
}
```

Hinweis Anweisungen wie ein Funktionsaufruf enden immer mit einem Semikolon.

Erste Idee der Objektorientierung

In einer objektorientierten Programmiersprache sind alle Methoden an bestimmte Objekte gebunden (daher der Begriff *objektorientiert*). Betrachten wir zum Beispiel das Objekt

4 Abkürzung für Methoden, die mit `print` beginnen, also `print()` und `println()`.

Radio. Ein Radio spielt Musik ab, wenn der Ein-Schalter betätigt wird und ein Sender und die Lautstärke eingestellt sind. Ein Radio bietet also bestimmte Dienste (Operationen) an, wie `Musik an/aus`, `lauter/leiser`. Zusätzlich hat ein Objekt auch noch einen Zustand. Es ist zum Beispiel die `Farbe` oder das `Baujahr`. Wichtig in objektorientierten Sprachen ist, dass die Operationen und Zustände immer (und da gibt es keine Ausnahmen) an Objekte beziehungsweise Klassen gebunden sind (mehr zu dieser Unterscheidung später). Der Aufruf einer Methode auf ein Objekt richtet die Anfrage genau an ein bestimmtes Objekt. Steht in einem Java-Programm einfach nur die Anweisung `lauter`, so weiß der Compiler nicht, wen er fragen soll. Was ist, wenn es auch noch einen Fernseher gibt? Aus diesem Grunde verbinden wir das Objekt, das etwas kann, mit der Operation. Ein Punkt trennt das Objekt von der Operation oder dem Zustand.

So gehört auch `println()` zu einem Objekt, welches die Bildschirmausgabe übernimmt. Dass eine Methode immer zu einem Objekt gehört, können wir auch an unserem eigenen Programm überprüfen. `main()` gehört zu der Klasse `Main`, die später ein Objekt bilden kann. Daher können wir in Java auch nicht einfach die Ausgabeanweisung schreiben. `println()` gehört zu einem Objekt mit dem Namen `out`. Dieses Objekt ist wiederum Teil der Klasse `System`. Wir können das vergleichen mit einem Aufruf zum Beispiel von

```
BRD.aktuellerBundeskanzler.fragen( "Wieso kassieren ARD" +
 " und ZDF jährlich 11 Mrd. Mark Rundfunkgebühren"+
 " und müssen davon nichts abführen?" );
```

Mehr zu diesen Aufrufen zu einem späterem Zeitpunkt. Das obige Beispiel macht aber jetzt schon deutlich, dass Strings mit dem Plus zusammengehängt werden können.

2.2.3 Die leere Anweisung

Die einfachste Anweisung besteht nur aus einem Semikolon und ist die *leere Anweisung*:

```
;
```

Sie wird verwendet, wenn die Sprachgrammatik eine Anweisung vorschreibt, aber in dem Programmablauf keine Anweisung vorkommen muss. So muss etwa hinter dem Schleifenkopf eine Anweisung folgen. Wir werden bei den Schleifen eine Anwendung der leeren Anweisung sehen.

2.2.4 Der Block

Ein Block innerhalb von Methoden oder statistischen Blöcken fasst eine Gruppe von Anweisungen zusammen, die hintereinander ausgeführt werden. Dazu werden die Anweisungen zwischen geschweiften Klammern geschrieben:

```
{
  Anweisung1;
  Anweisung2;
  ...
}
```

Ein Block kann überall dort verwendet werden, wo auch eine einzelne Anweisung stehen kann. Der neue Block hat jedoch eine Besonderheit für Variablen, denn er bildet einen lokalen Bereich für die darin befindlichen Anweisungen inklusive der Variablen. Blöcke können auch geschachtelt werden.

2.3 Elemente einer Programmiersprache

Wir haben einige Beispiele für Java-Programme gesehen und wollen nun über das Regelwerk, die Grammatik und Syntax sprechen. Wir wollen uns unter anderem über die Kodierung in Unicode, die Token und Bezeichner Gedanken machen.

2.3.1 Textkodierung durch Unicode-Zeichen

Die Algorithmen für Java-Programme bestehen aus einer Folge von Anweisungen und Unterprogrammen. In Anweisungen und Funktionsnamen werden Folgen von Zeichen als Bezeichner eingesetzt, die diese Bezeichner und Funktionen kennzeichnen. Wir müssen ihnen Namen geben, und dabei dürfen wir uns der Zeichen auf der Tastatur bedienen. Der Zeichenvorrat nennt sich auch *Lexikalik*.

Texte werden in Java durch 16 Bit lange *Unicode-Zeichen* kodiert. Der Unicode-Zeichensatz beinhaltet die ASCII-Zeichen nach ISO8859-1 (Latin-1), daher gehören alle gewöhnlichen Zeichen auch zum erweiterten Zeichensatz. Da mit 16 Bit etwa 65.000 Zeichen kodiert werden können, sind auch alle wichtigen Zeichensätze für andere Schriftsprachen kodiert. Eine angenehme Konsequenz ist, dass auch der Quellcode in der Landessprache programmiert werden kann. Deutsche Umlaute stellen demnach für den Compiler kein Hindernis dar.

Tipp Obwohl Java intern Unicode für alle Bezeichner einsetzt, ist es dennoch ungünstig, Klassennamen zu wählen, die Unicode-Zeichen enthalten. Das Problem liegt nicht in der Programmiersprache begründet, sondern im Dateisystem der meisten Betriebssysteme. Sie speichern die Namen oft im alten 8-Bit-ASCII-Zeichensatz ab.

Schreibweise für Unicode-Zeichen

Kaum ein Editor dürfte in der Lage sein, alle Unicode-Zeichen anzuzeigen. Beliebige Unicode-Zeichen lassen sich als `\uxxxx` schreiben, wobei `x` eine hexadezimale Ziffer ist – also 0..1, A..F beziehungsweise a..f. Diese Sequenzen können an beliebiger Stelle eingesetzt werden. So können wir anstatt eines Anführungszeichens alternativ `\u0027` schreiben, und dies wird vom Compiler als gleichwertig angesehen. Das Unicode-Zeichen `\uffff` ist nicht definiert und kann daher bei Zeichenketten als Ende-Symbol verwendet werden. Unicode-Zeichen für deutsche Sonderzeichen sind folgende.

Zeichen	Unicode
Ä,ä	`\u00c4`, `\u00e4`
Ö, ö	`\u00d6`, `\u00f6`
Ü,ü	`\u00dc`, `\u00fc`
ß	`\u00df`

Tabelle 2.1 Deutsche Sonderzeichen in Unicode

Anzeige der Unicode-Zeichen

Die Darstellung der Zeichen ist unter den meisten Plattformen noch ein großes Problem. Die Unterstützung für die Standardzeichen des ASCII-Alphabets ist dabei weniger ein Problem als die Sonderzeichen, die der Unicode-Standard definiert. Unter ihnen zum Beispiel der beliebte Smiley :-), der als Unicode `\u263A` (WHITE SMILING FACE) und `\u2369` (WHITE FROWNING FACE) :-(definiert ist. Das Euro-Zeichen ist unter `\u20ac` zu finden.

Tipp Sofern die Sonderzeichen und Umlaute sich auf der Tastatur befinden, sollten keine Unicode-Kodierungen Verwendung finden. Der Autor von Quelltext sollte seine Leser nicht zwingen, eine Unicode-Tabelle zur Hand zu haben. Die Alternativdarstellung lohnt sich daher nur, wenn der Programmtext bewusst unleserlich gemacht werden soll.

Ein Versuch, den Smiley auf die Standardausgabe zu drucken, scheitert oft an der Fähigkeit des Terminals beziehungsweise der Shell. Hier ist eine spezielle Shell nötig, die aber bei den meisten Systemen erst noch in der Entwicklung ist. Und auch bei grafischen Oberflächen ist die Integration noch mangelhaft. Es wird Aufgabe der Betriebssystementwickler bleiben, dies zu ändern.[5]

2.3.2 Unicode-Tabellen unter Windows

Unter Windows legt Microsoft das nützliche Programm CHARMAP.EXE für eine Zeichentabelle bei, mit der jede Schriftart auf ihre installierten Zeichen untersucht werden kann. Praktischerweise zeigt die Zeichentabelle auch gleich die Position in der Unicode-Tabelle an.

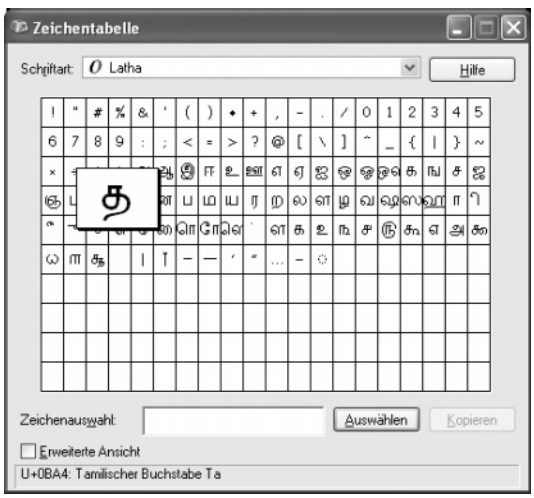


Abbildung 2.1 Zeichentabelle unter Windows XP

5 Mit veränderten Dateiströmen lässt sich dies etwas in den Griff bekommen. So lässt sich beispielsweise mit einem speziellen `OutputStream`-Objekt eine Konvertierung für die Windows-NT-Shell vornehmen, so dass auch dort die Sonderzeichen erscheinen.

Unter der erweiterten Ansicht lassen sich zusätzlich auch noch **Unicode-Unterbereiche** auswählen, wie etwa **Währungszeichen** oder unterschiedliche Sprachen. Im Unterbereich **Latin** finden sich zum Beispiel die Zeichen aus der französischen (etwa C mit Cedille unter 00c7) und spanischen (n mit Tilde unter 00F1) Schrift und bei **Allgemeinen Interpunktionszeichen** findet sich das umgedrehte (invertierte) Fragezeichen bei 00BF.

2.3.3 Bezeichner

Für Variablen (und damit Konstanten), Methoden, Klassen und Schnittstellen werden *Bezeichner* vergeben, die die entsprechenden Bausteine anschließend im Programm identifizieren. Unter Variablen sind dann Daten verfügbar, Methoden sind die Prozeduren in objektorientierten Programmiersprachen und Klassen sind die Bausteine objektorientierter Programme.

Ein Bezeichner ist eine Folge von Zeichen, die fast beliebig lang sein kann (die Länge ist nur theoretisch festgelegt). Die Zeichen sind Elemente aus dem gesamten Unicode-Zeichensatz, und jedes Zeichen ist für die Identifikation wichtig. Das heißt, ein Bezeichner, der 100 Zeichen lang ist, muss auch immer mit allen 100 Zeichen korrekt angegeben werden. Manche C- und FORTRAN-Compiler sind in dieser Hinsicht etwas großzügiger und bewerten nur die ersten Stellen. Jeder Bezeichner muss mit einem Unicode-Buchstaben beginnen. Dies sind Unicode-Zeichen zum Beispiel aus dem Bereich »A« bis »Z« und »a« bis »z« (nicht beschränkt auf lateinische Zeichen) sowie »_« und »$«.[6] Nach dem ersten Buchstaben können neben den Buchstaben auch Ziffern folgen. Dass der Unterstrich mittlerweile mit zu den Buchstaben zählt, ist nicht weiter verwunderlich, doch dass das Dollarzeichen mitgezählt wird, ist schon erstaunlich. Sun erklärt den Einsatz einfach damit, dass diese beiden Zeichen »aus historischen Gründen« mit aufgenommen wurden. Eine sinnvollere Erklärung ist, dies mit der Verwendung von maschinengeneriertem Code zu erklären. Es bleibt noch einmal zu erwähnen, dass zwischen Groß-/Kleinschreibung unterschieden wird.

Die folgende Tabelle listet einige gültige und ungültige Bezeichner auf.

Gültige Bezeichner	Ungültige Bezeichner
`mami`	`2und2macht4`
`kulliReimtSichAufUlli`	`class`
`IchWeißIchMussAndréAnrufen`	`hose gewaschen`
`raphaelIstLieb`	`hurtig!`

Tabelle 2.2 Beispiele für Bezeichner in Java

`mami` ist genau ein Bezeichner, der nur aus Alphazeichen besteht und daher korrekt ist. Auch `kulliReimtSichAufUlli` ist korrekt. Der Bezeichner zeigt zusätzlich die in Java übliche Bezeichnerbildung; denn besteht dieser aus mehreren einzelnen Wörtern, werden diese einfach ohne Leerzeichen hintereinander gesetzt, jedes Teilwort (außer dem ersten) beginnt jedoch dann mit einem Großbuchstaben. Leerzeichen sind in Bezeichnern nicht erlaubt, und daher ist auch `hose gewaschen` ungültig. Auch das Ausrufezeichen ist, wie viele

6 Ob ein Zeichen ein Buchstabe ist, verrät uns die Funktion `Character.isJavaLetter()`.

Sonderzeichen, ungültig. `IchWeißIchMussAndréAnrufen` ist jedoch wieder korrekt, auch wenn es ein Apostroph-é enthält. Treiben wir es weiter auf die Spitze, dann sehen wir einen gültigen Bezeichner, der nur aus griechischen Zeichen gebildet ist. Auch der erste Buchstabe ist ein Zeichen, anders als in `2und2macht4`. Und `class` ist ebenso ungültig, da der Name schon von Java belegt ist.

Hinweis In Java-Programmen bilden sich Bezeichnernamen oft aus zusammengesetzen Wörtern einer Beschreibung. Das bedeutet, dass in einem Satz wie »schönes Wetter heute« die Leerzeichen entfernt werden und die nach dem ersten Wort folgenden Worter mit Großbuchstaben beginnen. Damit wird aus dem Beispielsatz anschließend »schönesWetterHeute«. Sprachwissenschaftler nennen diese gemischte Groß- und Kleinschreibung *Binnenmajuskel*.

Die Tabelle im nächsten Abschnitt zeigt uns, welche Namen wir nicht verwenden können. Für `class` nehmen Programmierer als Ersatz gerne `clazz`.

2.3.4 Reservierte Schlüsselwörter

Bestimmte Wörter sind als Bezeichner nicht zulässig, da sie als *Schlüsselwörter* durch den Compiler besonders behandelt werden. Schlüsselwörter bestimmen die »Sprache« eines Compilers. Nachfolgende Zeichenfolgen sind Schlüsselwörter (beziehungsweise Literale im Fall von `true`, `false` und `null`)[7] und in Java daher nicht als Bezeichnernamen möglich:

abstract	assert	boolean	break	Byte
byvalue†	case	cast†	catch	char
class	const†	continue	default	do
double	else	extends	final	finally
float	for	future†	generic†	goto†
if	implements	import	instanceof	int
inner†	interface	long	native	new
null	operator†	outer†	package	private
protected	public	rest†	return	short
static	strictfp	super	switch	synchronized
this	throw	throws	transient	try
var†	void	volatile	while	

Tabelle 2.3 Reservierte Wörter in Java

Obwohl die mit † gekennzeichneten Wörter zur Zeit nicht von Java benutzt werden, können doch keine Variablen dieses Namens definiert werden.

7 Siehe dazu Kapitel 3.9 (Keywords) der Sprachdefinition unter `http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.htmlges`.

2.3.5 Token

Ein *Token* ist eine lexikalische Einheit, die dem Compiler die Bausteine des Programms liefert. Der Compiler erkennt an der Grammatik einer Sprache, welche Folgen von Zeichen ein Token bilden. Für Bezeichner heißt dies beispielsweise: Nimm die nächsten Zeichen, solange auf einen Buchstaben nur Buchstaben oder Ziffern folgen. Eine Zahl wie 1982 bildet zum Beispiel ein Token durch folgende Regel: Lese solange Ziffern, bis keine Ziffer mehr folgt. Bei Kommentaren bilden die Kombinationen `/*` und `*/` ein Token.

Problematisch wird es in einer Sprache immer dann, wenn der Compiler die Token nicht voneinander unterscheiden kann. Daher fügen wir *Trennzeichen* (engl. *White-Spaces*), auch *Wortzwischenräume* genannt, ein. Zu den Trennern zählen Leerzeichen, Tabulatoren, Zeilenvorschub- und Seitenvorschubzeichen. Außer als Trennzeichen haben diese Zeichen keine Bedeutung. Daher können sie in beliebiger Anzahl zwischen die Token gesetzt werden. Das heißt auch, beliebig viele Leerzeichen sind zwischen Token gültig. Und da wir damit nicht geizen müssen, können sie einen Programmabschnitt enorm verdeutlichen. Programme sind besser lesbar, wenn sie luftig formatiert sind.

2.3.6 Semantik

Die Syntax eines Java-Programms definiert die Token und bildet so das Vokabular. Richtig geschriebene Programme müssen aber dennoch nicht korrekt sein. Unter dem Begriff »Semantik« fassen wir daher die Bedeutung eines syntaktisch korrekten Programms zusammen. Die Semantik bestimmt, was das Programm macht. Die Abstraktionsreihenfolge ist also Lexikalik, Syntax und Semantik. Der Compiler durchläuft diese Schritte, bevor er den Bytecode erzeugen kann.

2.4 Datentypen

Java nutzt, wie es für imperative Programmiersprachen typisch ist, Variablen zum Ablegen von Daten. Eine Variable ist ein reservierter Speicherbereich und belegt eine feste Anzahl von Bytes. Alle Variablen (und auch Ausdrücke) haben einen *Typ*, der zur Übersetzungszeit bekannt ist. Der Typ wird auch *Datentyp* genannt, da eine Variable einen Datenwert, auch *Datum* genannt, hält. Für jeden Typ lässt sich die Speichergröße berechnen. Beispiele für Datentypen sind: Ganzzahlen, Fließkommazahlen, Wahrheitswerte, Zeichen und Objekte. Da jede Variable einen festen Datentyp hat und diesen nicht mehr ändern kann, zählt Java zu den strengtypisierten Sprachen.[8] Der Datentyp erlaubt dem Übersetzer auch, die Daten im Speicher nach bestimmten Regeln zu behandeln. Wenn wir einen Speicherauszug lesen und dort die Bitinformationen 01011010, 11010010, 01010011, 10100010 finden, ist uns auch nicht klar, ob dies nun vier Buchstaben sind, eine Fließkommazahl oder eine Ganzzahl.

Die grobe Richtung

Die Datentypen in Java zerfallen in zwei Kategorien: *primitive Typen* und *Referenztypen* (auch *Klassentypen*). Die primitiven Typen sind die eingebauten Datentypen, die nicht als Objekte verwaltet werden. Referenztypen sind genau das, was noch übrig bleibt. Es gibt

8 Im Gegensatz dazu steht Smalltalk. In Smalltalk sind zuerst einmal alles Objekte, und diese haben keinen Typ. Die Operationen werden erst zur Laufzeit an die Objekte gebunden.

aber auch Sprachen, die keine primitiven Datentypen besitzen. Als Beispiel sei noch einmal auf das in der Fußnote angesprochene Smalltalk verwiesen.

Warum Sun sich für diese Teilung entschieden hat, lässt sich mit zwei Gründen erklären:

- Viele Programmierer kennen Syntax und Semantik von C(++) und anderen imperativen Programmiersprachen. Auf die neue Sprache Java zu wechseln fällt leicht, und das objektorientierte Denken aus C++ hilft, sich auf der Insel zurechtzufinden.
- Der andere Grund ist die Geschwindigkeit der häufig vorkommenden elementaren Rechenoperationen. Ein Compiler kann einerseits viele Optimierungen vornehmen, und das ist bei einer Sprache, die in der Industrie eingesetzt wird, ein wichtiges Kriterium. Die Klassen liefern andererseits die optimale Unterstützung für eine Wiederverwendung und skalierbare Produkte.

Wir werden uns im Folgenden erst mit primitiven Datentypen beschäftigen. Referenzen werden nur dann eingesetzt, wenn Objekte ins Spiel kommen. Dies dauert jedoch noch etwas.

2.4.1 Primitive Datentypen

In Java gibt es einige eingebaute Datentypen für ganze Zahlen, Gleitkommazahlen nach IEEE 754, Zeichen und Wahrheitswerte. (Strings werden bevorzugt behandelt, sind aber lediglich Verweise auf Objekte.) Die folgende Tabelle gibt darüber einen Überblick. Anschließend betrachten wir jeden Datentyp präziser.

Schlüsselwort	Länge in Bytes	Belegung (Wertebereich)
`boolean`	1	`true` oder `false`
`char`	2	16-Bit Unicode Zeichen (0x0000...0xffff)
`byte`	1	-2^7 bis 2^7 – 1 (-128...127)
`short`	2	-2^15 bis 2^15 – 1 (-32768...32767)
`int`	4	-2^31 bis 2^31 – 1 (-2147483648...2147483647)
`long`	8	-2^63 bis 2^63 – 1 (-9223372036854775808...9223372036854775807)
`float`	4	1.40239846E-45f... 3.40282347E+38f
`double`	8	4.94065645841246544E-324... 1.79769131486231570E+308

Tabelle 2.4 Java-Datentypen und deren Wertebereiche

Zwei wesentliche Punkte zeichnen die primitiven Datentypen aus:

- Alle Datentypen haben eine festgesetzte Länge, die sich unter keinen Umständen ändert. Der Nachteil, dass der Programmierer die Länge eines Datentyps nicht kennt, besteht in Java nicht. In den Sprachen C/C++ bleibt dies immer unsicher, und die

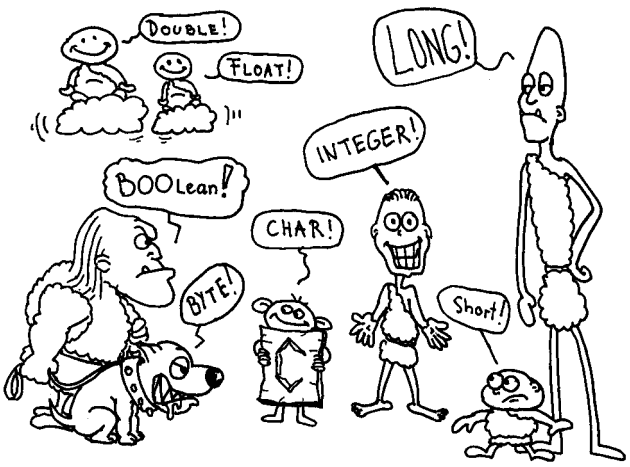
Umstellung auf 64-Bit-Maschinen bringt viele Probleme mit sich. Bei der Betrachtung der Auflistung fällt auf, dass `char` 16 Bit lang ist.

- Die numerischen Datentypen `byte`, `short`, `int` und `long` sind vorzeichenbehaftet, Fließkommazahlen sowieso. Dies ist leider nicht immer praktisch, aber wir müssen stets daran denken. Probleme gibt es, wenn wir einem Byte zum Beispiel den Wert 240 zuweisen wollen. Ein `char` ist im Prinzip ein vorzeichenloser Ganzzahltyp.

2.4.2 Wahrheitswerte

Der Datentyp `boolean` beschreibt einen Wahrheitswert, der entweder `true` oder `false` ist. Die Zeichenketten `true` und `false` sind reservierte Wörter und bilden so genannte *Literale*. Kein anderer Wert ist für Wahrheitswerte möglich.

Der boolesche Typ wird beispielsweise bei Bedingungen, Verzweigungen oder Schleifen benötigt.



2.4.3 Variablendeklarationen

Mit Variablen lassen sich Daten speichern, die vom Programm gelesen und geschrieben werden können. Um Variablen zu nutzen, müssen sie deklariert werden. Wir sprechen hier auch von der Definition[9] einer Variablen. Die Schreibweise einer Variablendeklaration ist immer die gleiche: Hinter dem Typnamen folgt der Name der Variablen.

```
Typname Variablenname;
```

Ein Variablenname (der dann Bezeichner ist) kann alle Buchstaben und Ziffern des Unicode-Zeichensatzes beinhalten, mit der Ausnahme, dass am Anfang einer Zeichenkette keine Ziffer stehen darf. Ebenfalls darf der Variablenname mit keinem reservierten Schlüsselwort identisch sein.

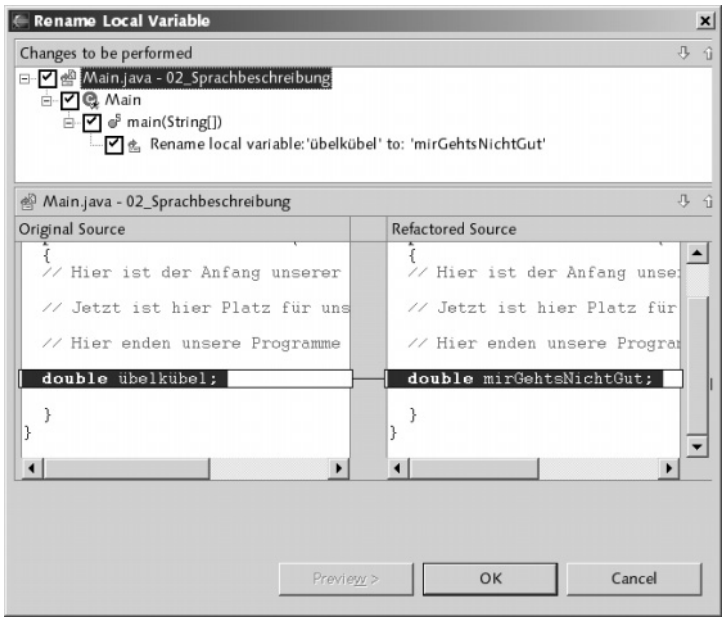
9 In C(++) bedeuten Definition und Deklaration etwas Verschiedenes. In Java kennen wir diesen Unterschied nicht und betrachten daher beide Begriffe als gleichwertig.

Hinweis Zwei Variablen ähnlicher Schreibweise, etwa `counter` und `counters`, führen schnell zu Verwirrung. Als Programmierer sollten wir uns konsistent an ein Namensschema halten.[10]

Die Variablendeklaration ist eine Anweisung und wird daher mit einem Semikolon abgeschlossen.

Beispiel Unicode-Sequenzen können vom Programmierer überall im Programm aufgenommen werden. Folgende Deklarationen mit den Bezeichnernamen sind daher gleich:

```
double übelkübel;
double \u00FCbelk\u00FCbel;
```



Ist ein Bezeichnername unglücklich gewählt, so lässt er sich ohne Probleme konsistent umbenennen. Dazu ist im Menü **Refactor**, **Rename** – oder auch kurz Alt+⇧+R – auszuwählen; der Cursor muss auf dem Bezeichner stehen. Ein optionaler Preview zeigt an, welche Änderungen die Umbenennung nach sich ziehen wird.

Werden mehrere Variablen gleichen Typs bestimmt, so können diese mit einem Komma getrennt werden. Eine Deklaration kann in jeden Block geschrieben werden:

```
Typname Variablenname1[, Variablenname2, ... ]
```

Schreiben wir ein einfaches Programm, welches eine Wahrheitsvariable definiert und zuweist. Die Variablenbelegung erscheint zusätzlich auf dem Bildschirm.

10 Eine Software wie Mathematica warnt vor Variablen mit fast identischem Namen.

Listing 2.3 FirstVariable.java

```
class FirstVariable
{
  public static void main( String args[] )
  {
    boolean trocken;

    trocken = true;
    System.out.print( "Ist die Socke trocken? " );
    System.out.println( trocken );
  }
}
```

Die Zeile `trocken = true` ist eine Zuweisung, die die Variable `trocken` mit einem Wert belegt. Sie ist ebenfalls eine Anweisung und wird mit einem Semikolon beendet. Steht auf der rechten Seite keine Variable, so steht dort ein Literal, eine Konstante, wie in unserem Fall `true`. Wir haben schon erwähnt, dass es für Wahrheitswerte nur die Literale `true` und `false` gibt.

2.4.4 Ganzzahlige Datentypen

Java stellt vier ganzzahlige Datentypen zur Verfügung: `byte`, `short`, `int` und `long`. Sie unterscheiden sich nur in der Länge, die jeweils 1, 2, 4 und 8 Byte umfasst. Die ganzzahligen Typen (lassen wir `char` einmal außen vor) sind in Java immer vorzeichenbehaftet, und einen Modifizierer `unsigned` wie in C(++) gibt es nicht.[11]

Beispiel Variablendeklaration mit Wertinitialisierung

```
int schuhGröße;

int i = 1243, j = 01230, k = 0xcafebabe;
```

Den Variablen kann gleich bei der Definition ein Wert zugewiesen werden. Hinter einem Gleichheitszeichen wird der Wert geschrieben, der oft ein Literal ist. Eine Zuweisung gilt nur für immer genau eine Variable. Negative Zahlen werden durch Voranstellen eines Minuszeichens gebildet. Ein Pluszeichen für positive Zeichen ist optional.

Das hexadezimale und oktale Zahlensystem

Die Literale für Ganzzahlen lassen sich in drei unterschiedlichen Zahlensystemen angeben. Das natürlichste ist das Dezimalsystem, wie das Beispiel an der Variablen `i` zeigt. Die Literale bestehen aus den Ziffern »0« bis »9«. Zusätzlich existiert die Oktal- und Hexadezimalform, die die Zahlen zur Basis 8 und 16 schreiben. Ein oktaler Wert beginnt mit dem Präfix »0« und ein hexadezimaler Wert mit »0x«. Mit der Basis 8 werden nur die Ziffern »0« bis »7« für oktale Werte benötigt. Da zehn Ziffern für hexadezimale Zahlen nicht ausreichen, besteht eine Zahl zur Basis 16 zusätzlich aus den Buchstaben »a« bis »f« (beziehungsweise »A« bis »F«).

11 In Java bilden `long` und short einen eigenen Datentyp. Sie dienen nicht wie in C(++) als Modifizierer. Eine Deklaration wie `long int` ist also falsch.

Achtung Wer sich im kalifornischen Cupertino (unter anderem Apple-Hauptsitz) aufhält und dieses Buch liest, sollte es vermeiden, gut hörbar das Hexadezimalsystem rückwärts aufzuzählen. Das ist gesetzlich verboten!

Der Datentyp long

Ganzzahlen doppelter Größe werden mit einem »l« oder »L« am Ende versehen.

Beispiel Deklaration eines `long` mit angehängtem »`L`«

```
long l = 123456789098L, m = -1L, n = 0xC0B0L;
```

Betrachten wir folgende Zeile, so ist auf den ersten Blick kein Fehler zu erkennen:

```
System.out.println( 123456789012345 );
```

Der Übersetzungsvorgang fördert jedoch noch einmal zu Tage, dass alle Datentypen ohne explizite Größenangabe als `int` angenommen werden, das heißt, 32 Bit lang sind. Obige Zeile führt daher zu einem Compilerfehler, da die Zahl nicht im Wertebereich von `-2147483648` bis `2147483647` liegt. Java reserviert also nicht so viele Bits wie benötigt und wählt nicht automatisch den passenden Wertebereich. Er muss ausdrücklich angegeben werden. Um die Zahl `123456789012345` gültig ausgeben zu lassen, müssen wir schreiben:

```
System.out.println( 1234567890123451 );
```

Ersichtlich wird, dass ein kleines »l« sehr viel Ähnlichkeit mit der Ziffer Eins besitzt. Daher sollte bei Längenangaben immer ein großes »L« hinten angestellt werden.

```
System.out.println( 123456789012345L );
```

Allerdings ist das Compilerverhalten verwirrend, denn bei folgender Anweisung findet er auch automatisch die richtige Größe.

```
byte b = 12;
```

2.4.5 Die Fließkommazahlen

Java unterscheidet Fließkommazahlen einfacher Genauigkeit (`float`) und doppelter Genauigkeit (`double`). Die Datentypen sind im IEEE-754-Standard beschrieben und haben eine Länge von 4 Byte für `float` und 8 Byte für `double`.

Die Literale bestehen aus einem Vorkommateil, einem Dezimalpunkt (kein Komma) und einem Nachkommateil. Optional kann ein Exponent angegeben werden. Standardmäßig sind die Literale vom Typ `double`. Ein nachgestelltes »f« (oder »F«) zeigt an, dass es sich um ein `float` handelt. Vorkommateil und Exponent dürfen durch die Vorzeichen »+« oder »-« eingeleitet werden.

Beispiel Gültige Zuweisungen für Fließkommazahlen vom Typ `double` und `float`:

```
double pi = 3.1415, klein = .001, x = 3.00e+8;
float   y = 3.00E+8F;
```

Der Exponent kann entweder positiv oder negativ sein[12], muss aber eine Ganzzahl sein.

2.4.6 Zeichen

Der Datentyp `char` ist 2 Byte groß und nimmt ein Unicode-Zeichen auf. Ein `char` ist nicht vorzeichenbehaftet. Die Literale für Zeichen werden in einfache Hochkommata gesetzt. Spracheinsteiger verwechseln häufig die einfachen Hochkommata mit den Anführungszeichen der Zeichenketten (Strings). Die einfache Merkregel: Ein Zeichen – ein Hochkomma, mehrere Zeichen – zwei Hochkommata (Gänsefüßchen).

Beispiel Korrekte Hochkommata für Zeichen und Zeichenketten:

```
char   c = 'a';
String s = "Heut' schon gebeckert?";
```

Escape-Sequenzen/Fluchtsymbole

Für spezielle Zeichen stehen Escape-Sequenzen[13] zur Verfügung, die so nicht direkt als Zeichen dargestellt werden können.

Zeichen	Bedeutung
`\b`	Rückschritt (Backspace)
`\n`	Zeilenschaltung (Newline)
`\f`	Seitenumbruch (Formfeed)
`\r`	Wagenrücklauf (Carriage return)
`\t`	Horizontaler Tabulator
`\"`	Doppeltes Anführungszeichen
`\'`	Einfaches Anführungszeichen
`\\`	Backslash

Tabelle 2.5 Escape-Sequenzen

Beispiel Zeichenvariablen mit Initialwerten und Sonderzeichen:

```
char          a = 'a',
    singlequote = '\'',
        newline = '\n',
```

Die Fluchtsymbole sind für Zeichenketten die gleichen. Auch dort können bestimmte Zeichen mit Escape-Sequenzen dargestellt werden.

12 LOGO verwendet für negative Exponenten den Buchstaben N anstelle des E. In Java bleibt das E mit einem folgenden unären Plus- oder Minus-Zeichen.

13 Nicht alle aus C stammenden Escape-Sequenzen finden sich auch in Java wieder. Es gibt kein '\a' (Alert), '\v' (vertikaler Tabulator), '\?' (Ausrufezeichen) und kein '\x', was eine hexadezimale Zahl einleitet (dafür lässt sich in Java \uXXXX nutzen).

Beispiel `String s = "Er fragte: \"Wer lispelt wie Katja Burkard?\"";`

2.4.7 Die Typanpassung (das Casting)

Möglicherweise kommt es vor, dass Datentypen konvertiert werden müssen. Dies nennt sich *Typanpassung* (engl. *Typecast*) oder auch *casten*. Java unterscheidet zwei Arten der Typanpassung:

- **Automatische Typanpassung**
  Daten eines kleineren Datentyps werden automatisch dem größeren angepasst.
- **Explizite Typanpassung**
  Ein größerer Typ kann einem kleineren Typ nur mit Verlust von Informationen zugewiesen werden.

Automatische Anpassung der Größe

Werte der Datentypen `byte` und `short` werden bei Rechenoperationen automatisch in den Datentyp `int` umgewandelt. Ist ein Operand vom Datentyp `long`, dann werden alle Operanden auf `long` erweitert. Wird aber `short` oder `byte` als Ergebnis verlangt, dann ist dieses durch einen expliziten Typecast anzugeben und nur die niederwertigen Bits des Ergebniswerts werden übergeben. Folgende Typumwandlungen sind ohne Informationsverlust möglich:

Von Typ	In Typ
`byte`	`short, char, int, long, float, double`
`short`	`int, long, float, double`
`char`	`int, long, float, double`
`int`	`long, float, double`
`long`	`float, double`
`float`	`double`

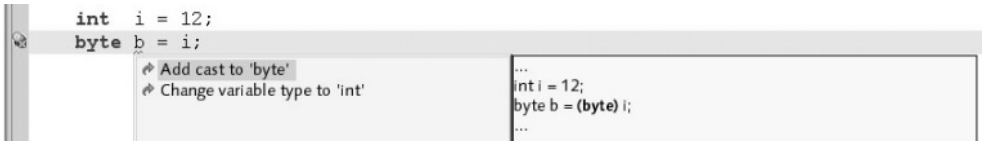
Tabelle 2.6 Zuweisungen ohne Informationsverlust

Explizite Typanpassung

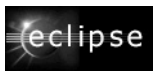
Der gewünschte Typ für eine Typanpassung wird vor den umzuwandelnden Datentyp geschrieben. Der gewollte Datentyp ist geklammert.

Beispiel Umwandlung einer Fließkommazahl in eine Ganzzahl:

```
int n = (int) 3.1415;
```



Passt der Typ eines Ausdrucks nicht, lässt er sich mit Strg+1 korrigieren.



Eine Typumwandlung hat eine sehr hohe Priorität. Daher muss der Ausdruck gegebenenfalls geklammert werden.

Beispiel Die Zuweisung an `n` verfehlt das Ziel.

```
int n = (int) 1.0315 + 2.1;
int m = (int)(1.0315 + 2.1);       // das ist korrekt
```

Typumwandlung von Fließkommazahlen zu Ganzzahlen

Bei der expliziten Typumwandlung von `double` und `float` in einen Ganzzahltyp kann es selbstverständlich zum Verlust von Genauigkeit kommen sowie zur Einschränkung des Wertebereichs. Bei der Konvertierung von Fließkommazahlen verwendet Java eine Rundung gegen Null.

Beispiel Zahlen, die bei der Konvertierung die Rundung nach Null aufzeigen

```
double w = +12.34;
double x = +67.89;
double y = -12.34;
double z = -67.89;
System.out.println( (int) w );  // 12
System.out.println( (int) x );  // 67
System.out.println( (int) y );  // -12
System.out.println( (int) z );  // -67
```

Bei der Konvertierung eines größeren Ganzzahltyps in einen kleineren werden einfach die oberen Bits abgeschnitten. Eine Anpassung des Vorzeichens findet nicht statt.

```
int i = 123456789;
int j = -123456;
System.out.println( (short) i );  // -13035
System.out.println( (short) j );  // 7616
```

short und char

Ein `short` hat wie ein `char` eine Länge von 16 Bit. Doch diese Umwandlung ist nicht ohne ausdrückliche Konvertierung möglich. Das liegt am Vorzeichen von `short`. Zeichen sind per Definition immer ohne Vorzeichen. Würde ein `char` mit einem gesetzten höchstwertigen letzten Bit in ein `short` konvertiert, käme eine negative Zahl heraus. Ebenso wäre, wenn ein `short` eine negative Zahl bezeichnet, das oberste Bit im `char` gesetzt, was unerwünscht ist. Die ausdrückliche Umwandlung erzeugt immer nur positive Zahlen.

Der Verlust bei der Typumwandlung von `char` nach `short` tritt etwa bei der Han-Zeichenkodierung für chinesische, japanische oder koreanische Zeichen auf. Denn dort ist im Unicode das erste Bit gesetzt, welches bei der Umwandlung in ein `short` dem nicht gesetzten Vorzeichenbit weichen muss.

Typanpassungen von int und char

Die Methode `printXXX()` reagiert auf die Typen `char` und `int`, und eine Typumwandlung führt zu der gewünschten Ausgabe.

```
int  c1 = 65;
char c2 = 'A';

System.out.println( c1 );            // 65
System.out.println( (int)c2 );       // 65
System.out.println( (char)c1 );      // A
System.out.println( c2 );            // A
System.out.println( (char)(c1+1) );  // B
System.out.println( c2+1 );          // 66
```

Einen Ganzzahlwert in einem `int` können wir als Zeichen ausgeben, genauso wie eine `char`-Variable als Zahlenwert. Wir sollten beachten, dass eine mathematische Operation auf `char`-Typen zu einem `int` führt.

Probleme bei Zuweisungen

Leider ist die Typanpassung nicht ganz so einleuchtend, wie folgendes Beispiel zeigt.

Listing 2.4 AutoConvert.java

```
public class AutoConvert
{
  public static void main( String args[] )
  {
    int   i1 = 1, i2 = 2, i3;
    long  l1 = 1, l2 = 2, l3;
    short s1 = 1, s2 = 2, s3;
    byte  b1 = 1, b2 = 2, b3;
    i3 = i1 + i2;              // das ist noch OK
    l3 = l1 + l2;
//    s3 = s1 + s2;              // Compilerfehler!
//    b3 = b1 + b2;
    s3 = (short) ( s1 + s2 );  // das ist wieder OK
    b3 = (byte)  ( b1 + b2 );
  }
}
```

Dies ist auf den ersten Blick paradox. Es ist nicht möglich, ohne explizite Typumwandlung zwei `short`- oder `byte`-Zahlen zu addieren. Das Verhalten des Übersetzers lässt sich mit der automatischen Anpassung erklären. Wenn Ganzzahl-Ausdrücke vom Typ kleiner `int` mit einem Operator verbunden werden, passt der Compiler eigenmächtig den Typ auf `int` an.

Die Addition der beiden Zahlen arbeitet also nicht mit `short`- oder `byte`-Werten, sondern mit `int`-Werten. So werden auch Überläufe korrekt behandelt.

Bei der Zuweisung wird dies zum Problem. Denn dann steht auf der rechten Seite ein `int` und auf der linken Seite der kleinere Typ `byte` oder `short`. Nun muss der Compiler meckern, da Zahlen abgeschnitten werden könnten. Mit der ausdrücklichen Typumwandlung erzwingen wir diese Konvertierung und akzeptieren ein paar fehlende Bits. Diese Eigenart ist insofern verwunderlich, als dass doch auch ein `int` nur dann zu einem `long` erweitert wird, wenn einer der Operanden eines Ausdrucks vom Typ `long` ist.

Materialverlust durch Überläufe

Überläufe bei Berechnungen können zu schwer wiegenden Fehlern führen, so wie beim Absturz der Ariane 5 am 4. Juni 1996 genau 36.7 Sekunden nach dem Start. Die europäische Raumfahrtbehörde European Space Agency (ESA) startete von Französisch-Guyana aus eine unbemannte Rakete mit vier Satelliten an Bord, die 40 Sekunden nach dem Start explodierte. Glücklicherweise kamen keine Menschen ums Leben, doch der materielle Schaden belief sich auf etwa 500 Millionen US-Dollar. In dem Projekt steckten zusätzlich Entwicklungskosten von etwa 7 Milliarden US-Dollar. Grund für den Absturz war ein Rundungsfehler, der durch die Umwandlung einer 64-Bit-Fließkommazahl (die horizontale Geschwindigkeit) in eine vorzeichenbehaftete 16-Bit-Ganzzahl auftrat. Die Zahl war leider größer als 2^{15}, und die Umwandlung war nicht gesichert, da die Programmierer diesen Zahlenbereich nicht angenommen hatten. Die Konsequenz war, dass das Lenksystem zusammenbrach und die Selbstzerstörung ausgelöst wurde, da die Triebwerke abzubrechen drohten. Das wirklich Dumme an dieser Geschichte ist, dass die Software nicht unbedingt für den Flug notwendig war und nur den Startvorbereitungen diente. Im Fall einer Unterbrechung während des Countdowns hätte dann das Programm schnell abgebrochen werden können. Ungünstig war, dass der Programmteil unverändert durch Wiederverwendung per Copy-and-Paste aus der Ariane-4-Software kopiert worden war, die Ariane 5 aber schneller flog.

2.4.8 Lokale Variablen, Blöcke und Sichtbarkeit

In jedem Block und auch in jeder Klasse[14] können Variablen deklariert werden. Globale Variablen, die für alle Funktionen und Klassen sichtbar sind, gibt es in Java nicht. (Eine globale Variable müsste in einer Klasse definiert werden, die dann alle Klassen übernehmen.)

Sichtbarkeit

Jede Variable hat einen *Geltungsbereich, auch Gültigkeitsbereich genannt* (engl. *scope*). Sie ist nur in dem Block gültig, in dem sie definiert wurde. In dem Block ist die Variable lokal.

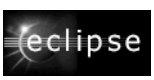
Beispiel Da ein Block immer mit geschweiften Klammern angegeben wird, erzeugen wir durch folgende Funktionen Blöcke, die einen weiteren inneren Block besitzen. Somit sind Blöcke ineinander geschachtelt.

14 Die so genannten Objektvariablen oder Klassenvariablen, doch dazu später mehr.

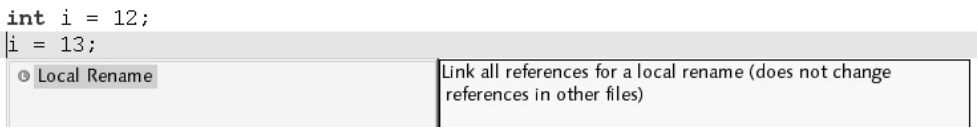
```
void foo()
{
  int i;
  {
    int j;                // j gilt nur in dem Block
    j = 1;
  }
//  j = 2;                // Funktioniert auskommentiert nicht
}
void bar()
{
  int i, k;             // i hat mit oberem i nichts zu tun
  {
//    int k;             // Das würde nicht gehen!
  }
}
```

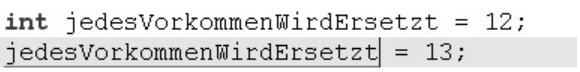
Zu jeder Zeit können Blöcke definiert werden. Außerhalb des Blocks sind deklarierte Variablen nicht sichtbar. Nach Abschluss des inneren Blocks, der `j` deklariert, ist ein Zugriff auf `j` nicht mehr möglich; auf `i` ist der Zugriff weiterhin erlaubt. Falls Objekte im Block angelegt wurden, wird der GC diese wieder freigeben, falls keine zusätzliche Referenz besteht.

Variablennamen können innerhalb eines Blocks nicht genauso gewählt werden wie lokale Variablennamen eines äußeren Blocks oder wie die Namen für die Parameter einer Funktion. Das zeigt zum Beispiel die Definition der Variablen `k`. Obwohl andere Programmiersprachen das erlauben, haben sich die Java-Sprachentwickler dagegen entschieden, um Fehlerquellen zu vermeiden.



Soll eine Variable in ihrem lokalen Kontext umbenannt werden, so gibt es neben dem Rename auch eine andere Möglichkeit. Dazu lässt sich auf der Variablen mit Strg+1 ein Popup-Fenster mit **Local Rename** öffnen. Der Bezeichner wird selektiert und lässt sich ändern. Gleichzeitig ändern sich alle Bezüge auf die Variable mit.





2.4.9 Initialisierung von lokalen Variablen

Während Objektvariablen automatisch mit einem Nullwert initialisiert werden, geschieht dies bei lokalen Variablen nicht. Das heißt, der Programmierer muss sich selbst um die Initialisierung kümmern.

Beispiel Häufig passieren Fehler bei falsch angewendeten bedingten Anweisungen, wie das folgende Programmsegment demonstriert.

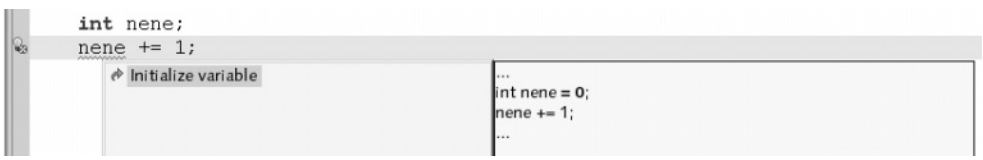
```
void test()
{
  int nene, williWurm;
  nene += 1;                    // Compilerfehler
  nene = 0; nene = nene + 1;
  if ( nene == 1 )
    williWurm = 2;
  williWurm = williWurm + 1;    // Compilerfehler
}
```

Die beiden lokalen Variablen `nene` und `williWurm` werden nicht automatisch mit Null initialisiert – so wie dies für Objektvariablen der Fall ist. So kommt es bei der Inkrementierung von `nene` zu einem Compilerfehler. Denn dazu ist erst ein Lesezugriff auf die Variable nötig, um anschließend den Wert `1` zu addieren. Der erste Zugriff muss aber eine Zuweisung sein. Das bedeutet, `nene=0` ist in Ordnung. Den Fehler würden wir auch bekommen, wenn wir in `System.out.println(nene)` die Variablenbelegung auslesen würden.

Oftmals gibt es jedoch bei Zuweisungen in bedingten Anweisungen Probleme. Da `williWurm` nur nach der `if`-Abfrage auf den Wert `2` gesetzt wird, wäre nur unter der Bedingung `nene` gleich `2` ein Lesezugriff auf `williWurm` möglich. Da diese Variable jedoch sonst vorher nicht gesetzt wurde, ergäbe sich das oben angesprochene Problem.

Ein Hinweis und Verbesserungsvorschlag, wenn eine lokale Variable nicht initialisiert ist.

eclipse



2.5 Ausdrücke

Beginnen wir mit mathematischen Ausdrücken, um dann die Schreibweise in Java zu ermitteln. Eine mathematische Formel, etwa der Ausdruck `-27*9`, besteht aus Operanden und Operatoren. Ein Operand ist eine Variable oder ein Literal. Im Fall einer Variablen wird der Wert der Variablen ausgelesen und danach die Berechnung gemacht.

Beispiel Ein Ausdruck mit Zuweisungen:

```
int i = 12, j;
j = i * 2;
```

Die Multiplikation berechnet das Produkt von 12 und 2 und speichert das Ergebnis in `j` ab. Von allen primitiven Variablen, die in dem Ausdruck vorkommen, wird also der Wert aus-

gelesen und in den Ausdruck eingesetzt.[15] Dies nennt sich auch *Wertoperation*, da der Wert der Variablen betrachtet wird und nicht ihr Speicherort oder gar ihr Variablenname.

Die Arten von Operatoren

Operatoren verknüpfen die Operanden. Ist ein Operator auf genau einem Operand definiert, so nennt er sich *unärer Operator* (oder *einstelliger Operator*). Das Minus (negatives Vorzeichen) vor einem Operand ist ein unärer Operator, da er für genau den folgenden Operanden gilt. Die üblichen Operatoren Plus, Minus, Mal und Division sind *binäre* (*zweistellige*) *Operatoren*. Es gibt auch einen Fragezeichenoperator für bedingte Ausdrücke, der dreistellig ist.

Ausdrücke

Ein Ausdruck ist eine besondere Form der Anweisung, denn ein Ausdruck ergibt bei der Auswertung ein Ergebnis (auch Resultat genannt). Dieses ist oft ein

- numerischer Typ (von arithmetischen Ausdrücken) oder ein
- Referenztyp (von einer Objekt-Allokation).

Operatoren erlauben die Verbindung von einzelnen Ausdrücken zu neuen Ausdrücken. Einige Operatoren sind aus der Schule bekannt, wie Addition, Vergleich, Zuweisung und Weitere. C(++)-Programmierer werden viele Freunde wiedererkennen.

2.5.1 Zuweisungsoperator und Verbundoperator

Der Zuweisungsoperator kopiert den Wert eines Ausdrucks der rechten Seite in die Variable der linken Seite. In Java wird eine Zuweisung mit dem Gleichheitszeichen = dargestellt.

```
int a;
a = 12*3;
```

Da eine Zuweisung keine Anweisung, sondern ein Ausdruck mit einem Wert ist, der einen Rückgabewert ergibt, kann die Zuweisung auch an jeder anderen Stelle eingesetzt werden, an der ein Ausdruck stehen darf. Die Zuweisung kann auch in einem Funktionsaufruf erfolgen:

```
System.out.println( a = 19 );
```

Auch Zuweisungen der Form

```
a = b = 0;
```

sind erlaubt und gleichbedeutend mit `b=0` und `a=b` beziehungsweise `a=(b=0)`.

Daran lässt sich ablesen, dass beim Zuweisungsoperator die Auswertung von rechts nach links erfolgt.

15 Es gibt Programmiersprachen, in denen werden Wertoperationen besonders gekennzeichnet. So etwa in LOGO. Eine Wertoperation schreibt sich mit einem Doppelpunkt vor der Variablen, etwa :X + :Y.

Beispiel Die Wahrheitsvariable `hatVorzeichen` soll dann `true` sein, wenn das Zeichen `vorzeichen` gleich dem Minus ist. Für Vergleiche dient der Operator ==:

```
boolean hatVorzeichen = (vorzeichen == '-');
```

Schön zu sehen an dem Beispiel ist die Auswertungsreihenfolge. Erst wird das Ergebnis des Vergleichs berechnet, und dieser Wahrheitswert wird anschließend in `hatVorzeichen` kopiert.

Verbundoperatoren

In Java können Zuweisungen mit numerischen (und auch bitweisen, aber dazu später) Operatoren kombiniert werden. Für einen Operator # in dem Ausdruck `a = a # (b)` gilt die Abkürzung durch einen *Verbundoperator* `a #= b`. So addiert der Ausdruck `a += 2` zur Variable `a` 2 hinzu. Der Rückgabewert ist die um 2 erhöhte Variable `a`.

Falls es sich bei der rechten Seite um einen komplexeren Ausdruck handelt, wird dieser nur einmal ausgewertet. Dies ist wichtig bei Methodenaufrufen, die Seiteneffekte besitzen.

Beispiel Wir profitieren auch bei einem Feldzugriff von Verbundoperationen, da der Zugriff auf das Feldelement nur einmal stattfindet.

```
feld[2*i+j] = feld[2*i+j] + 1;
```

Leichter zu lesen ist die folgende Anweisung:

```
feld[2*i+j] += 1;
```

In der Langform `a = a # (b)` ist die Klammerung wichtig, denn bei dem Ausdruck `a*=3+5` gilt `a=a*(3+5)` und durch die Punkt-vor-Strich-Regelung nicht `a=a*3+5`.

2.5.2 Präfix- oder Postfix-Inkrement und -Dekrement

Erhöhen/Erniedrigen von Variablen ist eine sehr häufige Operation, wofür die Entwickler in der Vorgängersprache C auch einen Operator spendiert haben. Die praktischen Operatoren ++ und `--` kürzen die Programmzeilen zum Inkrement und Dekrement ab.

```
i++;     // Abkürzung für i = i + 1
j--;     //               j = j - 1
```

Eine lokale Variable muss allerdings vorher initialisiert sein, da ein Lesezugriff vor einem Schreibzugriff stattfindet.

Vorher oder nachher

Die beiden Operatoren sind glücklicherweise Ausdrücke, so dass sie einen Wert liefern. Es macht jedoch einen feinen Unterschied, wo dieser Operator platziert wird. Es gibt ihn nämlich in zwei Varianten: vor der Variablen und dahinter. Steht das Inkrement vor der Variablen, sprechen wir von *Prä-Inkrement/Prä-Dekrement*, steht es dahinter von *Post-Inkrement/Post-Dekrement*: kurz auch *Präfix/Postfix*. Nutzen wir einen Präfix-Operator, so wird die Variable erst erhöht beziehungsweise erniedrigt und dann der Wert geliefert. Neben der Wertrückgabe gibt es eine Veränderung der Variablen.

Beispiel Präfix/Postfix in einer Ausgabeanweisung:

```
int i = 10, j = 20;
System.out.println( ++i );       // 11
System.out.println( --j );       // 19
System.out.println( i );         // 11
System.out.println( j );         // 19
```

Wir erkennen hier, dass der Wert erst erhöht wird und anschließend in die Berechnung eingeht. Deutlicher ist der Unterschied beim Postfix:

```
System.out.println( i++ );       // 10
System.out.println( j-- );       // 20
System.out.println( i );         // 11
System.out.println( j );         // 19
```

Das bedeutet, der Wert wird im Ausdruck verwendet und erst anschließend erhöht. Wir bekommen mit dem Präfix den Ausdruck nach der Operation und mit dem Postfix den Ausdruck davor.

Den Post-Inkrement finden wir auch im Namen der Programmiersprache C++. Es soll ausdrücken, dass es C-mit-eins-drauf ist, also ein verbessertes C. Mit dem Wissen über den Postfix-Operator ist klar, dass wir erst einen Zugriff haben und dann die Erhöhung stattfindet – also C++ ist auch nur C, und der Vorteil kommt später. (Einer der Entwickler von Java, Bill Joy, hat einmal Java als C++-- beschrieben. Er meinte damit C++ ohne die schwer zu pflegenden Eigenschaften.)

Einige Besonderheiten

Wir wollen uns abschließend noch mit einer Kuriosität des Post-Inkrements und Prä-Inkrements beschäftigen, die nicht nachahmenswert ist:

```
a = 2;
a = ++a;       // a = 3

b = 2;
b = b++;       // b = 2
```

Im ersten Fall bekommen wir den Wert 3 und im zweiten Fall 2. Der erste Fall überrascht nicht. Denn `a=++a` erhöht den Wert 2 um 1, und anschließend wird 3 der Variablen `a` zugewiesen. Bei `b` ist es raffinierter. Der Wert von `b` ist 2, und dieser Wert wird intern vermerkt. Anschließend erhöht `b++` die Variable `b`. Doch die Zuweisung setzt `b` auf den gemerkten Wert, der 2 war. Also ist `b=2`.

2.5.3 Unäres Minus und Plus

Die binären Operatoren sitzen zwischen zwei Operanden, während sich ein unärer Operator genau einen Operanden vornimmt. Das unäre Minus (Operator zur Vorzeichenumkehr) etwa dreht das Vorzeichen des Operanden um. So wird aus einem positiven Wert ein negativer und aus einem negativen ein positiver. Das unäre Plus ist eigentlich unnötig; die Entwickler haben es jedoch aus Symmetriegründen mit eingeführt.

Minus und Plus sitzen direkt vor dem Operator, und der Compiler weiß selbstständig, ob dies unär oder binär ist. Er hat es leicht, wenn typische Ausdrücke wie

```
a = -2;
```

geschrieben werden.

Beispiel Der Compiler erkennt auch folgende Konstruktion:

```
int i = - - - 2 + - + 3;
```

Dies ergibt den Wert `-5`.

Achtung! Der Compiler erkennt einen Ausdruck wie `---2+-+3` nicht an, da die zusammenhängenden Minuszeichen als Inkrement erkannt werden und nicht als unärer Operator.

2.5.4 Arithmetische Operatoren

Ein arithmetischer Operator verknüpft die Operanden mit den Operatoren Addition (+), Subtraktion (-), Multiplikation (*) und Division (/). Zusätzlich gibt es einen Modulo-Operator (auch Restwertoperator genannt), der den bei der Division verbleibenden Rest betrachtet. Das Zeichen für den Modulo-Operator ist %. Er ist für ganzzahlige Werte sowie für Fließkommazahlen definiert. Die arithmetischen Operatoren sind binär, und auf der linken und rechten Seite sind die Typen numerisch. Der Ergebnistyp ist ebenfalls numerisch. Bei unterschiedlichen Datentypgrößen werden vor der Anwendung der Operation alle Operanden auf den größten vorkommenden Typ gebracht. Anschließend wird die Operation ausgeführt, und der Ergebnistyp entspricht dem umfassenderen Typ.

Der Divisionsoperator

Der binäre Operator »/« bildet den Quotienten aus Dividend und Divisor. Auf der linken Seite steht der Dividend und auf der rechten der Divisor. Die Division ist für Ganzzahlen und für Fließkommazahlen definiert. Bei der Ganzzahldivision wird zur Null hin gerundet. Schon in der Schulmathematik war die Division durch Null nicht definiert. Führen wir eine Ganzzahldivision mit dem Divisor 0 durch, so bestraft uns Java mit einer `ArithmeticException`. Bei Fließkommazahlen verläuft dies anders. Eine Division durch 0 liefert meistens bei +/- unendlich eine `NaN`; außer bei 0.0/0.0. Ein `NaN` steht für *Not-A-Number* und wird vom Prozessor erzeugt, falls er eine mathematische Operation wie die Division durch Null nicht durchführen kann.

Der Modulo-Operator %

Bei einer Ganzzahldivision kann es passieren, dass wir einen Rest bekommen. So geht die Division 9/2 nicht auf. Der Rest ist 1. In Java sowie in C(++) ist es der Modulo-Operator (engl. *remainder operator*), der uns diese Zahl liefert. Somit ist 9%2 gleich 1. Im Gegensatz zu C(++)[16] erlaubt der Modulo-Operator in Java auch Fließkommazahlen, und die Operanden können negativ sein. Die Sprachdefinition von C(++) schreibt bei der Division und

16 Wir müssten in C(++) die Funktion `fmod()` benutzen.

beim Modulo mit negativen Zahlen keine Berechnungsmethode vor. In Java richten sich die Division und der Modulo nach einer einfachen Formel: int*(a/b)*b + (a%b) = a*[17]

Hinweis In Java sind Modulo (%), Inkrement (++) und Dekrement (--) für alle numerischen Datentypen erlaubt.[18]

Beispiel Die Gleichung ist erfüllt, wenn wir etwa a = 10 und b = 3 wählen. Es gilt: int(10/3) = 3. 10%3 ergibt 1. Dann ergeben 3*3+1 = 10.

Aus dieser Gleichung folgt, dass beim Modulo das Ergebnis nur dann negativ ist, wenn der Dividend negativ ist; er ist nur dann positiv, wenn der Dividend positiv ist. Es ist leicht einzusehen, dass das Ergebnis der Modulo-Operation immer echt kleiner ist als der Wert des Divisors. Wir haben den gleichen Fall wie bei der Ganzzahldivision, dass ein Divisor mit dem Wert 0 eine `ArithmeticException` auslöst.

Beispiel Unterschiedliche Vorzeichen beim Modulo-Operator:

Listing 2.5 ModuloDivDemo.java

```
class ModuloDivDemo
{
  public static void main( String args[] )
  {
    System.out.println( "5%3 = " + (5%3) );       // 2
    System.out.println( "5/3 = " + (5/3) );       // 1
    System.out.println( "5%-3 = " + (5%-3) );     // 2
    System.out.println( "5/-3 = " + (5/-3) );     // -1
    System.out.println( "-5%3 = " + (-5%3) );     // -2
    System.out.println( "-5/3 = " + (-5/3) );     // -1
    System.out.println( "-5%-3 = " + (-5%-3) );  // -2
    System.out.println( "-5/-3 = " + (-5/-3) );  // 1
  }
}
```

Modulo für Fließkommazahlen

Über die oben genannte Formel können wir auch bei Fließkommazahlen das Ergebnis einer Modulo-Operation leicht berechnen. Dabei muss beachtet werden, dass sich der Operator nicht so wie unter IEEE 754 verhält. Denn diese Norm schreibt vor, dass die Modulo-Operation den Rest von einer rundenden Division berechnet und nicht von einer abschneidenden Division. So wäre das Verhalten nicht analog zum Modulo bei Ganzzahlen. Java definiert das Modulo jedoch bei Fließkommazahlen genauso wie das Modulo auf Ganzzahlen. Wünschen wir ein Modulo-Verhalten wie es IEEE 754 vorschreibt, so können wir immer noch die Bibliotheksfunktion `Math.IEEEremainder()` verwenden.

17 In C sind sie nur für Ganzzahlen definiert.

18 Es gibt Programmiersprachen, wie APL, die keine Vorrangregeln kennen. Sie werten die Ausdrücke streng von rechts nach links oder umgekehrt aus.

Auch bei der Modulo-Operation bei Fließkommazahlen werden wir niemals eine Exception erwarten. Eventuelle Fehler werden, wie im IEEE-Standard beschrieben, mit `NaN` angegeben. Ein Überlauf oder Unterlauf kann zwar passieren, aber nicht geprüft werden.

Beispiel Modulo bei Fließkommazahlen:

```
5.0%3.0 = 2.0
5.0%-3.0 = 2.0
-5.0%3.0 = -2.0
-5.0%-3.0 = -2.0
```

Anwendung des Modulo-Operators

Im Folgenden wollen wir eine Anwendung des Modulo-Operators kennen lernen, denn dieser wird häufig dafür verwendet, eine einfache Überprüfung vorzunehmen. Bei einer eingegebenen Nummer wird dann einfach der Modulo-Wert zu einer fest vorgegebenen Zahl berechnet, und ist dieser zum Beispiel 0, so ist die Nummer eine gültige Codenummer. Erstaunlicherweise gibt es Firmen, die tatsächlich nach diesem einfachen Kodierungsverfahren ihre Software schützen, vorne dabei ist Microsoft mit Windows 95 und Windows NT 4.0. Nachdem die Software installiert ist, wird der Benutzer aufgefordert, einen CD-Key einzugeben. Es werden von den älteren Softwarepaketen drei unterschiedliche Schlüssellängen verwendet.

- **Normale Version mit 10 Stellen: xxx-NNNNNNN**
  Die ersten drei Stellen werden nicht geprüft. Die sieben letzten Ziffern müssen eine Quersumme ergeben, die durch 7 teilbar ist, also für die x % 7 = 0 gilt. So ist 1234567 eine gültige Zahl, da 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28 und 28 % 7 = 0 ist.
- **OEM Version: xxxxx-OEM-NNNNNNN-xxxxx**
  Die ersten acht und die letzten fünf Stellen werden nicht geprüft. Die in der Mitte liegenden restlichen sieben Stellen werden nach dem gleichen Verfahren wie in der normalen Version geprüft.
- **Neuer CD-Schlüssel: xxxx-NNNNNNN**
  Die ersten vier Ziffern steuern, ob es sich um eine Vollversion (0401) oder eine Update-Version (0502) handelt. Die restlichen sieben Ziffern sind wieder Modulo 7 zu nehmen.

Das Verfahren ist mehr als einfach. Die Hintergründe sollen natürlich nicht zum Installieren illegal erworbener Software führen. Dies verstößt selbstverständlich gegen das Urheberrecht! Diese Variante darf höchstens dann angewendet werden, wenn die Produkt-ID für die lizenzierte Software verloren ging!

Rundungsfehler

Prinzipiell sollten Anweisungen wie 1.1 – 0.1 immer 1.0 ergeben, doch interne Rundungsfehler bei der Darstellung treten auf und lassen das Ergebnis von Berechnung zu Berechnung immer ungenauer werden. Ein besonders ungünstiger Fehler trat 1994 beim Pentium Prozessor im Divisionsalgorithmus Radix-4 SRT auf, ohne dass der Programmierer der Schuldige war.

```
double x, y, z;

x = 4195835.0;
y = 3145727.0;
z = x - (x/y) * y;

System.out.println( z );
```

Ein fehlerhafter Prozessor liefert hier 256, obwohl laut Rechenregel das Ergebnis 0 sein muss. Laut Intel sollte für einen normalen Benutzer (Spieler, Softwareentwickler, Surfer?) der Fehler nur alle 27.000 Jahre auftauchen. Glück für die meisten. Eine Studie von IBM errechnet eine Fehlerhäufigkeit von einmal in 24 Tagen. Alles in allem hat Intel die CPUs zurückgenommen, über 400 Millionen US-Dollar verloren und spät den Kopf gerade noch aus der Schlinge gezogen.

Die meisten Rundungsfehler resultieren aber daher, dass endliche Dezimalbrüche im Rechner als Näherungswerte für periodische Binärbrüche repräsentiert werden müssen. 0,1 entspricht einer periodischen Mantisse im IEEE-Format.

2.5.5 Die relationalen Operatoren

Relationale Operatoren vergleichen Ausdrücke miteinander und geben einen Wahrheitswert vom Typ `boolean` zurück. Ein anderes Wort für relationaler Operator ist *Vergleichsoperator*. Die von Java zur Verfügung gestellten Operatoren sind Größer (`>`), Kleiner (`<`), Test auf Gleichheit (`==`) und Ungleichheit (`!=`) sowie die Verbindung zu einem Größer-gleich (`>=`) beziehungsweise Kleiner-gleich (`<=`).

Ebenso wie arithmetische Operatoren passen die relationalen Operatoren ihre Operanden an einen gemeinsamen Typ an. Handelt es sich bei den Typen um Referenztypen, so sind nur die Vergleichsoperatoren `==` und `!=` erlaubt.

Verwechslungsprobleme durch == und =

Die Verwendung des relationalen Operators `==` und der Zuweisung `=` führt bei Einsteigern oft zu Problemen, da die Mathematik für beide immer nur ein Gleichheitszeichen kennt. Glücklicherweise ist das Problem in Java nicht so drastisch wie beispielsweise in C(++), da die Typen der Operatoren unterschiedlich sind. Der Vergleichsoperator ergibt immer nur den Rückgabewert `boolean`. Zuweisungen von numerischen Typen ergeben jedoch wieder einen numerischen Typ. Es kann also kein Problem wie das folgende geben:

```
int a = 10, b = 11;
boolean result1 = ( a = b );        // Compilerfehler
boolean result2 = ( a == b );       // Das ist OK
```

2.5.6 Logische Operatoren

Mit logischen Operatoren werden Wahrheitswerte nach definierten Mustern verknüpft. Logische Operatoren operieren nur auf `boolean`-Typen, andere Typen führen zu Compilerfehlern. Java bietet die Operatoren Und (`&&`), Oder (`||`), Xor (`^`) und Nicht (`!`) an. Xor ist eine Operation, die genau dann falsch zurückgibt, wenn entweder beide Operatoren wahr oder beide falsch sind. Sind sie unterschiedlich, so ist das Ergebnis wahr.

Kurzschlussoperatoren

Eine Besonderheit sind die *Kurzschlussoperatoren* (engl. *short-circuit-operator*) && beziehungsweise || für Und und Oder. In der Regel wird ein logischer Ausdruck nur dann weiter ausgewertet, wenn er das Schlussergebnis noch beeinflussen kann. Sonst optimiert der Compiler die Programme zum Beispiel bei zwei Operanden:

- **Und:** Ist einer der beiden Ausdrücke falsch, so kann der Ausdruck schon nicht mehr wahr werden. Das Ergebnis ist falsch.
- **Oder:** Ist mindestens einer der Ausdrücke schon wahr, so ist auch der gesamte Ausdruck wahr.

Es ist aber in einigen Fällen gewünscht, dass alle Teilausdrücke ausgewertet werden, insbesondere wenn Funktionen Seiteneffekte bewirken. Daher führt Java zusätzliche Nicht-Kurzschlussoperatoren | und & ein, die in einem komplexen Ausdruck alle Teilausdrücke auswerten. Für das ausschließende Oder Xor (Operator ^) kann es keinen Kurzschlussoperator geben, da immer beide Operanden ausgewertet werden müssen, bevor das Ergebnis feststeht.

Beispiel In dem ersten Ausdruck wird die Methode `foo()` nicht aufgerufen, im zweiten schon.

```
b = true || foo();   // foo() wird nicht aufgerufen.
b = false & foo();   // foo() wird aufgerufen.
```

2.5.7 Reihenfolge und Rang der Operatoren in der Auswertungsreihenfolge

Aus der Schule ist der Spruch »Punktrechnung geht vor Strichrechnung« bekannt, so dass Ausdrücke der Art

```
1 + 2 * 3
```

zu 7 und nicht zu 9 ausgewertet werden. In den meisten Programmiersprachen gibt es eine Unzahl von Operatoren neben Plus und Mal, die alle ihre eigenen Vorrangregeln besitzen. Der Multiplikationsoperator besitzt zum Beispiel eine höhere Vorrangregel (kurz Rang) und damit eine andere Auswertungsreihenfolge als der Plus-Operator.[19]

Beispiel Zur Umwandlung einer Temperatur von Fahrenheit in Celsius wird von dem Wert in Fahrenheit 32 abgezogen und das Ergebnis mit 5/9 multipliziert:

```
celsius = fahrenheit - 32 * 5 / 9;
```

Die erste Idee ist aber leider falsch, denn hier berechnet der Compiler 32*5/9. Das Ergebnis 17 wird von Fahrenheit abgezogen, was keine gültige Umrechnung ist. Richtig ist Folgendes:

```
celsius = ( fahrenheit - 32 ) * 5 / 9;
```

19 Es gibt Programmiersprachen, wie APL, die keine Vorrangregeln kennen. Sie werten die Ausdrücke streng von rechts nach links oder umgekehrt aus.

Die Rechenregeln für Mal vor Plus kann sich jeder noch leicht merken. Komplizierter ist die Auswertung bei den zahlreichen Operatoren, die seltener im Programm vorkommen.

Beispiel Wie ist die Auswertung bei dem nächsten Ausdruck?

```
boolean A = false,
        B = false,
        C = true;
System.out.println( A && B || C );
```

Gilt, dass entweder `A && B` oder `C` wahr sein müssen oder etwa `A` und `B || C`? Das Ergebnis fällt unterschiedlich aus. Entweder ist es `true` oder `false`.

Für derlei Feinheiten gibt es zwei Lösungen: entweder in einer Tabelle mit Vorrangregeln nachschlagen oder auf diese Ungenauigkeiten verzichten.

Operator	Rang	Typ	Beschreibung
`++, --`	1	arithmetisch	Inkrement und Dekrement
`+, -`	1	arithmetisch	unäres Plus und Minus
`~`	1	integral	bitweises Komplement
`!`	1	boolean	logisches Komplement
(Typ)	1	jedes	Cast
`*, /, %`	2	arithmetisch	Multiplikation, Division, Rest
`+, -`	3	arithmetisch	Addition und Subtraktion
`+`	3	String	String-Konkatenation
`<<`	4	integral	Shift links
`>>`	4	integral	Shift rechts m. Vorzeichenerweiterung
`>>>`	4	integral	Shift rechts o. Vorzeichenerweiterung
`<, <=, >, >=`	5	arithmetisch	numerische Vergleiche
`instanceof`	5	Objekt	Typvergleich
`==, !=`	6	primitiv	Gleich-/Ungleichheit von Werten
`==, !=`	6	Objekt	Gleich-/Ungleichheit von Referenzen
`&`	7	integral	bitweises Und
`&`	7	boolean	logisches Und
`^`	8	integral	bitweises Xor

Tabelle 2.7 Operatoren mit Rangordnung in Java

Operator	Rang	Typ	Beschreibung
^	8	boolean	logisches Xor
|	9	integral	bitweises Oder
|	9	boolean	logisches Oder
&&	10	boolean	logisches konditionales Und, Kurzschluss
||	11	boolean	logisches konditionales Oder, Kurzschluss
?:	12	alles	Bedingungsoperator
=	13	jede	Zuweisung
*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	14	jede	Zuweisung mit Operation

Tabelle 2.7 Operatoren mit Rangordnung in Java (Forts.)

Die Tabelle lehrt uns, dass im Beispiel `A && B || C` das Und stärker als das Oder bindet, also der Wert mit der Belegung `A=false`, `B=false`, `C=true` zu `true` ausgewertet wird. Vermutlich gibt es Programmierer, die dies wissen oder eine Tabelle mit Rangordnungen immer am Monitor kleben haben. Aber beim Durchlesen von fremdem Code ist es nicht schön, immer wieder die Tabelle konsultieren zu müssen, die verrät, ob nun das binäre Xor oder das binäre Und stärker binden.

Tipp Alle Ausdrücke, die über die einfache Regel »Punktrechung geht vor Strichrechnung« hinausgehen, sollten geklammert werden. Da die unären Operatoren ebenfalls sehr stark binden, kann eine Klammerung wegfallen.

Beispiel Bei den Operatoren +, * gilt die mathematische Kommutativität und Assoziativität. Das heißt, die Operanden können prinzipiell umgestellt werden, und das Ergebnis sollte davon nicht beeinträchtigt sein. Bei der Division gilt das nicht.

```
A / B / C
```

Der Ausdruck wird von links nach rechts ausgewertet, und zwar als `(A/B)/C`. Hier sind Klammern angemessen. Denn würde der Compiler den Ausdruck zu `A/(B/C)` auswerten, käme es einem `A*C/B` gleich.

Die mathematische Assoziativität gilt bei Gleitkommazahlen natürlich nicht, da diese nicht ohne Rechenfehler ablaufen. Daher gilt eine Auswertung von links nach rechts.

2.5.8 Was C(++)-Programmierer vermissen könnten

Da es in Java keine Pointer-Operationen gibt, existiert das Operatorzeichen zur Referenzierung (`&`) und Dereferenzierung (`*`) nicht. Ebenso ist ein `sizeof` unnötig, da das Laufzeitsystem und der Compiler immer die Größe von Klassen kennen beziehungsweise die primitiven Datentypen immer eine feste Länge haben. Eine abgeschwächte Version vom Kommaoperator ist in Java nur im Kopf von `for`-Schleifen erlaubt.

2.6 Bedingte Anweisungen oder Fallunterscheidungen

Verzweigungen dienen in einer Programmiersprache dazu, Programmteile unter bestimmten Bedingungen auszuführen. Java bietet eine `if`- und `if`/`else`-Anweisung sowie die `switch`-Anweisung zum Ausführen verschiedener Programmteile. Obwohl ein Sprung mit `goto` nicht möglich ist, besitzt Java eine spezielle Sprungvariante: `continue` und `break` mit definierten Sprungzielen.

2.6.1 Die if-Anweisung

Die `if`-Anweisung besteht aus dem Schlüsselwort `if`, dem zwingend ein Ausdruck mit dem Typ `boolean` in Klammern folgt. Es folgt eine Anweisung, die oft eine Blockanweisung ist.

```
if ( Ausdruck )
  Anweisung
```

Die Abarbeitung der Anweisung hängt nun vom Ausdruck ab. Ist das Ergebnis des Ausdrucks wahr`(true)`, so wird die Anweisung ausgeführt. Ist das Ergebnis des Ausdrucks falsch `(false)`, so wird mit der ersten Anweisung nach der `if`-Anweisung fortgefahren.

Beispiel Ein Relationenvergleich

```
if ( x < y )
  System.out.println( "x ist kleiner als y" );
```

Im Gegensatz zu C(++) muss der Testausdruck in der `if`-Anweisung (übrigens auch in den folgenden Schleifen) vom Typ `boolean` sein. In C(++) wird ein numerischer Ausdruck als wahr bewertet, wenn das Ergebnis des Ausdrucks ungleich 0 ist.

Betrachten wir in einer `if`-Anweisung den Vergleich, ob ein Objekt existiert. Dann ist dies mit `null` zu vergleichen.[20] Die Referenz auf das Objekt steht in der Variablen `ref`.

```
if ( ref != null )
  ...
```

if-Anfragen und Blöcke

Hinter dem `if` und der Bedingung erwartet der Compiler eine Anweisung. Wenn wir jedoch mehrere Anweisungen in Abhängigkeit von der Bedingung ausführen wollen, so müssen wir einen Block verwenden, denn andernfalls ordnet der Compiler nur die nächstfolgende Anweisung der Fallunterscheidung zu, auch wenn mehrere Anweisungen optisch

20 Und nicht einfach `if (ref)` wie in C(++).

abgesetzt sind.[21] Dies ist eine große Gefahr für Programmierer, die optisch Zusammenhänge schaffen wollen, die in Wirklichkeit nicht existieren.

Beispiel Eine `if`-Anweisung soll testen, ob die Variable `y` den Wert 0 hat. In dem Fall soll sie die Variable `x` auf 0 setzen und zusätzlich auf dem Bildschirm »`Null`« anzeigen. Zunächst die semantisch falsche Variante:

```
if ( y == 0 )
  x = 0;
  System.out.println( "Null" );
```

Sie ist semantisch falsch, da unabhängig von `y` immer eine Ausgabe erscheint. Der Compiler interpretiert die Zeilen in folgendem Zusammenhang:

```
if ( y == 0 )
  x = 0;

System.out.println( "Null" );
```

Damit das Programm korrekt wird, müssen wir einen Block verwenden und die Anweisungen zusammensetzen.

Beispiel Ein korrekt geklammerter Ausdruck:

```
if ( y == 0 ) {
  x = 0;
  System.out.println( "Null" );
}
```

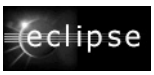
Zusammengesetzte Bedingungen

Unsere bisherigen Abfragen waren sehr einfach, jedoch kommen in der Praxis viel komplexere Bedingungen vor. Dafür werden häufig die logischen Operatoren `&&`, `||` beziehungsweise `!` verwendet. Wenn wir etwa testen wollen, ob eine Zahl `x` entweder gleich 7 oder größer gleich 10 ist, schreiben wir die zusammengesetzte Bedingung

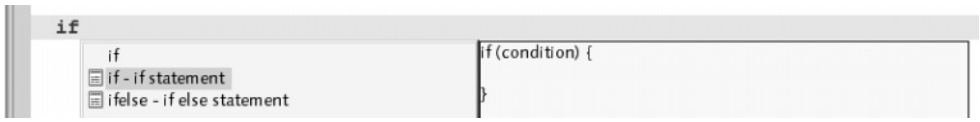
```
if ( x == 7 || x >= 10 )
  ...
```

Sind die logisch verknüpften Ausdrücke komplexer, so sollten zur Unterstützung der Lesbarkeit die einzelnen Bedingungen in Klammern gesetzt werden, da nicht jeder sofort die Tabelle mit den Vorrangregeln für die Operatoren im Kopf hat.

21 In der Programmiersprache Python bestimmt die Einrückung die Zugehörigkeit.



`if` und Strg+␣ bietet an, eine `if`-Anweisung mit Block anzulegen.



2.6.2 Die Alternative wählen mit einer if/else-Anweisung

Neben der einseitigen Alternative existiert die zweiseitige Alternative. Das optionale Schlüsselwort `else` veranlasst die Ausführung der alternativen Anweisung, wenn der Test falsch ist:

```
if ( Ausdruck )
  Anweisung1
else
  Anweisung2
```

Falls der Ausdruck wahr ist, wird die Anweisung 1 ausgeführt, andernfalls Anweisung 2. Somit ist sichergestellt, dass in jedem Fall eine Anweisung ausgeführt wird.

```
if ( x < y )
  System.out.println( "x ist echt kleiner als y." );
else
  System.out.println( "x ist größer oder gleich y." );
```

Dangling-Else-Problem

Bei Verzweigungen mit `else` gibt es ein bekanntes Problem, welches *Dangling-Else-Problem* genannt wird. Zu welcher Anweisung gehört das folgende `else`?

```
if ( Ausdruck1 )
  if ( Ausdruck2 )
    Anweisung1;
else
  Anweisung2;
```

Die Einrückung suggeriert, dass das `else` die Alternative zur ersten `if`-Anweisung ist. Dies ist aber nicht richtig. Die Semantik von Java (und auch fast aller anderen Programmiersprachen) ist so definiert, dass das `else` zum innersten `if` gehört. Daher lässt sich nur der Programmiertipp geben, die `if`-Anweisungen zu klammern:

```
if ( Ausdruck1 )
{
  if ( Ausdruck2 )
  {
    Anweisung1;
  }
}
else
{
  Anweisung2;
}
```

So kann eine Verwechslung gar nicht erst auftreten.

Beispiel Wenn das `else` immer zum innersten `if` gehört, und das ist nicht erwünscht, können wir, wie gerade gezeigt, mit geschweiften Klammern arbeiten oder auch eine leere Anweisung im `else`-Zweig zufügen:

```
if ( x >= 0 )
  if ( x != 0 )
    System.out.println( "x echt größer Null" );
  else
    ; // x ist gleich Null
else
  System.out.println( "x echt kleiner Null" );
```

Das böse Semikolon

An dieser Stelle ist ein Hinweis angebracht. Ein Programmieranfänger schreibt gerne hinter die schließende Klammer der `if`-Anweisung ein Semikolon. Das führt zu einer ganz anderen Ausführungsfolge. Ein Beispiel:

```
int alter = 29;
if ( alter < 0 ) ;
  System.out.println( "Aha, noch im Mutterleib" );
if ( alter > 150 ) ;
  System.out.println( "Aha, ein neuer Moses" );
```

Das Semikolon führt dazu, dass die leere Anweisung in Abhängigkeit von der Bedienung ausgeführt wird und unabhängig vom Inhalt der Variable `alter` immer die Ausgabe »Aha, noch im Mutterleib« erzeugt. Das ist sicherlich nicht beabsichtigt.

Folgen hinter einer `if`-Anweisung zwei Anweisungen, die nicht durch eine Blockanweisung zusammengefasst sind, dann wird die eine folgende `else`-Anweisung als Fehler bemängelt, da der zugehörige `if`-Zweig fehlt. Der Grund ist, dass der `if`-Zweig nach der ersten Anweisung ohne `else` zu Ende ist.

```
int alter = 29;
if ( alter < 0 ) ;
  System.out.println( "Aha, noch im Mutterleib" );
else ( alter > 150 ) ;
  System.out.println( "Aha, ein neuer Moses" );
```

Das führt zu der Fehlermeldung `'else' without 'if'`.

Mehrfachverzweigung oder auch geschachtelte Alternativen

`if`-Anweisungen zur Programmführung kommen sehr häufig in Programmen vor, und noch häufiger ist es, eine Variable auf einen bestimmten Wert zu prüfen. Dazu werden `if`- und `if`/`else`-Anweisung gerne geschachtelt. Wenn eine Variable einem Wert entspricht, dann wird eine Anweisung ausgeführt, sonst wird die Variable mit einem anderen Wert getestet und so weiter.

Dieser Ansatz ist sehr umständlich und kostet zudem noch Rechenzeit, da in jedem Fall drei Bedingungen geprüft werden. Wenn also `x` größer Null ist, werden dennoch zwei Ver-

gleiche gemacht. Wir schachteln daher in einer kleinen Programmverbesserung die Alternativen und arbeiten dann mit einer Abfolge von sequenziell abhängigen Alternativen:[22]

```
if ( x > 0 )
  signum = 1;
else
  if ( x < 0 )
    signum = -1;
  else
    signum = 0;
```

Jetzt werden nur noch so viele Bedingungen geprüft wie zur Entscheidung notwendig sind. Die eingerückten Verzweigungen nennen sich auch *angehäufte if-Anweisungen* oder *if-Kaskade*, da jede `else`-Anweisung ihrerseits weitere `if`-Anweisungen enthält, bis alle Abfragen gemacht sind.

Beispiel Kaskadierte `if`-Anweisungen:

```
if ( monat == 4 )
  tage = 30;
else
  if ( monat == 6 )
    tage = 30;
  else
    if ( monat == 9 )
      tage = 30;
    else
      if ( monat == 11 )
        tage = 30;
      else
        if ( monat == 2 )
          if ( schaltjahr )
            tage = 29;
          else
            tage = 28;
        else
          tage = 31;
```

2.6.3 Die switch-Anweisung bietet die Alternative

In Java[23] gibt es eine Kurzform für speziell gebaute, angehäufte `if`-Anweisungen – die Anweisung mit `switch` und `case`:

```
switch ( Ausdruck )
{
  case Konstante:
    Anweisungen
}
```

22 In der Programmiersprache Euphoria (die Webseite *http://www.rapideuphoria.com/* wirbt mit safe und sexy - Huuh) heißt das einfach: return (x>0)-(x<0).

23 Und auch in C(++)

Die switch-Anweisung ist eine einfache Form der Mehrfachverzweigung. Sie vergleicht nacheinander den Ausdruck hinter dem switch (ein primitiver Typ wie byte, char, short oder int) mit jedem einzelnen Fallwert. Alle Fallwerte müssen unterschiedlich sein. Stimmt der Ausdruck mit der Konstanten überein, so wird die Anweisung beziehungsweise die Anweisungen hinter der Sprungmarke ausgeführt.

Hinweis Eine Einschränkung der switch-Anweisung besteht darin, dass die Tests und die Konstanten nur auf den primitiven Datentyp int beschränkt sind.

Es können also keine größeren Typen wie long oder Fließkommazahlen wie float beziehungsweise double oder gar Objekte benutzt werden. Als Alternative bleiben nur angehäufte if-Anweisungen. Dies ist auch der einzige Weg, um Bereiche abzudecken.

Alles andere abdecken mit default

Gibt es keine Übereinstimmung mit einer Konstanten, so lässt sich optional die Sprungmarke default einsetzen:

```
switch ( ausdruck )
{
  case Konstante:
    Anweisungen
  ...
  default:
}
```

Ohne Übereinstimmung mit einem konkreten Ziel geht die Abarbeitung des Programmcodes hinter default weiter. default kann auch zwischen den Konstanten eingesetzt werden.

Beispiel Ein Taschenrechner mit Alternative

```
switch ( op )
{
  case '+':   // addiere
    break;

  case '-':   // subtrahiere
    break;

  case '*':   // multipliziere
    break;

  case '/':   // dividiere
    break;

  default:
    System.err.println( "Operand nicht definiert!" );
}
```

switch hat Durchfall

Bisher haben wir in die letzte Zeile eine `break`-Anweisung gesetzt. Ohne ein `break` würden nach einer Übereinstimmung alle nachfolgenden Anweisungen ausgeführt. Sie laufen somit in einen neuen Abschnitt herein bis ein `break` oder das Ende von `switch` erreicht ist. Da dies vergleichbar mit einem Spielzeug ist, bei dem Kugeln von oben nach unten durchfallen, nennt sich dieses auch *Fall-Through*. Ein häufiger Programmierfehler ist, das `break` zu vergessen, und daher sollte ein beabsichtigter Fall-Through immer als Kommentar angegeben werden.

Beispiel Über dieses Durchfallen ist es möglich, bei unterschiedlichen Werten immer die gleiche Anweisung ausführen zu lassen.

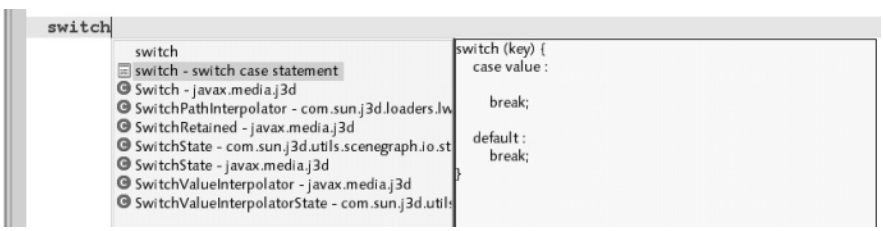
```
switch ( buchstabe )
{
  case 'a':                    // Durchfallen
  case 'e':
  case 'i': case 'o': case 'u':
    vokal = true;
    break;

  default:
    vokal = false;
}
```

In dem Beispiel bestimmt eine `case`-Anweisung, ob die Variable `buchstabe` ein Vokal ist. Fünf `case`-Anweisungen decken jeweils einen Buchstaben ab. Stimmt die Variable mit einer Konstanten überein, so »fällt« der Interpreter in den Programmcode der Zuweisung. Dieses Durchfallen über die `case`-Zweige ist praktisch, so wie es unser Programmcode für das Ist-Vokal-Problem zeigt. Der erste `case`-Zweig setzt die boolesche Variable `vokal` bei einem Vokal auf wahr. Tritt die Bedingung nicht ein, so weist die Anweisung im `default`-Teil der Variablen `vokal` den Wert falsch zu.

Hinweis Obwohl ein fehlendes `break` zu lästigen Programmierfehlern führt, haben die Java-Entwickler dieses Verhalten vom syntaktischen Vorgänger C übernommen. Eine interessante Lösung wäre gewesen, das Verhalten genau umzudrehen und das Durchfallen explizit einzufordern, zum Beispiel mit einem Schlüsselwort.

eclipse

`switch` und Strg+␣ bietet an, ein Grundgerüst für eine `switch` Fallunterscheidung anzulegen.



2.7 Schleifen

Schleifen dienen dazu, bestimmte Anweisungen immer wieder abzuarbeiten. Zu einer Schleife gehören die Schleifenbedingung und der Rumpf. Die Schleifenbedingung entscheidet darüber, unter welcher Bedingung die Wiederholung ausgeführt wird. Sie muss ein boolescher Ausdruck sein. In Abhängigkeit von der Schleifenbedingung kann der Rumpf mehrmals ausgeführt werden. Dazu wird bei jedem Schleifendurchgang die Schleifenbedingung geprüft. Das Ergebnis entscheidet, ob der Rumpf ein weiteres Mal durchlaufen (`true`) oder die Schleife beendet wird (`false`).

2.7.1 Die while-Schleife

Die `while`-Schleife ist eine abweisende Schleife, da sie vor jedem Schleifeneintritt die Schleifenbedingung prüft:

```
while ( Ausdruck )
  Anweisung
```

Vor jedem Schleifendurchgang wird der Ausdruck neu ausgewertet und ist das Ergebnis `true`, so wird der Rumpf ausgeführt. Die Schleife ist beendet, wenn das Ergebnis `false` ist. Ist die Bedingung schon vor dem ersten Eintritt in den Rumpf nicht wahr, so wird der Rumpf erst gar nicht durchlaufen. Der Typ der Bedingung muss `boolean` sein.

Wird innerhalb des Schleifenkopfs schon alles Interessante erledigt, so muss trotzdem eine Anweisung folgen. Dies ist der passende Einsatz für die leere Anweisung. Etwa

```
while ( leseWeiterBisZumEnde() )
  ;                             // Rumpf ist leer
```

Die Methode `leseWeiterBisZumEnde()` gibt `true` zurück, falls noch Zeichen gelesen werden können. Wenn der Rückgabewert `false` ist, so wird die Schleife beendet.

Da der Typ wiederum `boolean` sein muss, sehen die Anweisungen in Java im Gegensatz zu C(++) etwas präziser aus:

```
while ( i != 0 ) {              // und nicht while ( i ) wie in C(++)
  ...
}
```

Endlosschleifen

Ist die Bedingung einer `while`-Schleife immer wahr, dann handelt es sich um eine Endlosschleife. Die Konsequenz ist, dass die Schleife endlos wiederholt wird.

```
while ( true )
{
  // immer wieder und immer wieder
}
```

Aus dieser Endlosschleife lässt sich mittels `break` entkommen; das behandeln wir auf Seite Ausbruch planen mit break und Wiedereinstieg mit continueAusbruch planen mit break und Wiedereinstieg mit continueAusbruch planen mit break und Wiedereinstieg mit continue#. (Aber auch eine Exception oder `System.exit()` würden die Schleife beenden.)

2.7.2 Schleifenbedingungen und Vergleiche mit ==

Eine Schleifenabbruchbedingung kann ganz unterschiedlich aussehen. Beim Zählen ist es häufig der Vergleich auf einen Endwert. Oft steckt an dieser Stelle ein absoluter Vergleich mit ==, der aus zwei Gründen problematisch werden kann.

Lange, lange durchhalten

Beispiel Sehen wir uns das erste Problem an einigen Programmzeilen an:

```
int i = Wert;

while ( i != 9 )
  i++;
```

Ist der Wert der Variablen `i` kleiner als 9, so haben wir beim Zählen kein Problem, denn dann ist anschließend spätestens bei 9 Schluss. Ist der Wert allerdings echt größer als 9, so ist die Bedingung ebenso wahr, und der Schleifenrumpf wird ziemlich lange durchlaufen. Genau genommen so weit, bis wir durch einen Überlauf[24] wieder bei 0 beginnen und dann auch bei 9 landen. Die Absicht ist sicherlich eine andere gewesen. Die Schleife sollte nur solange zählen, wie `i` kleiner 9 ist und sonst nicht. Daher passt Folgendes besser:

```
int i = Wert;

while ( i < 9 )
  i++;
```

Jetzt rennt der Interpreter bei Zahlen größer 9 nicht endlos weiter, sondern stoppt die Schleife sofort.

Rechenungenauigkeiten

Das zweite Problem ergibt sich bei Gleitkommazahlen. Es ist sehr problematisch, echte Vergleiche zu fordern:

```
double d = 0.0;

while ( d != 1.0 )
{
  d += 0.1;
  System.out.println( d );
}
```

Lassen wir das Programmsegment laufen, so sehen wir, dass die Schleife hurtig über das Ziel hinausschießt:

```
0.1
0.2
0.30000000000000004
```

24 `Integer.MAX_VALUE + Integer.MAX_VALUE + 2` ist wieder gleich 0. Die Schleife muss also »nur« 4.294.967.296 Mal durchlaufen werden.

```
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
1.0999999999999999
1.2
1.3
```

... bis das Auge müde wird ...

Bei Fließkommawerten bietet sich daher immer an, mit den relationalen Operatoren `<` oder `>` zu arbeiten.

Eine zweite Möglichkeit neben dem echten Kleiner/Größer-Vergleich ist, eine erlaubte Abweichung zu definieren. Mathematiker bezeichnen die Abweichung von zwei Werten mit dem griechischen Kleinbuchstaben Epsilon. Wenn wir einen Vergleich von zwei Fließkommazahlen anstreben und bei einem Gleichheitsvergleich die Abweichung mit betrachten wollen, so schreiben wir einfach:

```
if ( Math.abs(a - b) <= epsilon )
  ...
```

Epsilon ist die erlaubte Abweichung. `Math.abs(x)` berechnet von einer Zahl `x` den Absolutwert.

2.7.3 Die do/while-Schleife

Dieser Schleifentyp ist eine annehmende Schleife, da die Schleifenbedingung erst nach jedem Schleifendurchgang geprüft wird. Bevor es zum ersten Test kommt, ist der Rumpf also schon einmal durchlaufen:

```
do
  Anweisung
while ( Ausdruck );                    // Bemerke das Semikolon
```

Es ist wichtig, auf das Semikolon hinter der `while`-Anweisung zu achten. Liefert die Bedingung ein `true`, so wird der Rumpf erneut ausgeführt. Andernfalls wird die Schleife beendet, und das Programm wird mit der nächsten Anweisung nach der Schleife fortgesetzt.

Beispiel Eine Zählschleife

```
int pos = 1;
do
{
  System.out.println( pos );
  pos++;
} while ( pos < 10 );
```

Äquivalenz einer while- und einer do/while-Schleife

Die `do`-Schleife wird seltener gebraucht als die `while`-Schleife. Dennoch lassen sich beide ineinander überführen. Zunächst der erste Fall. Wir ersetzen eine `while`-Schleife durch eine `do`/`while`-Schleife:

```
while ( Ausdruck )
  Anweisung
```

Führen wir uns noch einmal vor Augen, was hier passiert. In Abhängigkeit vom Ausdruck wird der Rumpf ausgeführt. Da zunächst ein Test kommt, wäre die `do`/`while`-Schleife schon eine Blockausführung weiter. So fragen wir in einem ersten Schritt mit einer `if`-Anweisung ab, ob die Bedingung wahr ist oder nicht. Wenn ja, dann lassen wir den Programmcode in einer `do`/`while`-Schleife abarbeiten. Die äquivalente `do`/`while`-Schleife sieht also wie folgt aus:

```
if ( Ausdruck )
  do
    Anweisung
  while ( Ausdruck ) ;
```

Nun der zweite Fall. Wir ersetzen die `do`/`while`-Schleife durch eine `while`-Schleife:

```
do
  Anweisung
while ( Ausdruck ) ;
```

Da zunächst die Anweisungen ausgeführt werden und anschließend der Test, schreiben wir für die `while`-Variante die Ausdrücke einfach vor den Test. So ist sichergestellt, dass diese zumindest einmal abgearbeitet wurden:

```
Anweisung
while ( Ausdruck )
  Anweisung
```

2.7.4 Die for-Schleife

Die `for`-Schleife ist eine spezielle Variante einer `while`-Schleife und wird typischerweise zum Zählen benutzt. Genauso wie `while`-Schleifen sind `for`-Schleifen abweisend, der Rumpf wird also erst dann ausgeführt, wenn die Bedingung wahr ist.

Beispiel Gebe die Zahlen von 1 bis 10 auf dem Bildschirm aus:

```
for ( int i = 1; i <= 10; i++ )
  System.out.println( i );
```

Eine genauere Betrachtung der Schleife zeigt die unterschiedlichen Segmente:

- **Initialisierung der Schleife**
  Der erste Teil der `for`-Schleife ist ein Ausdruck wie `i=1`, der vor der Durchführung der Schleife genau einmal ausgeführt wird. Dann wird das Ergebnis verworfen. Tritt in der Auswertung ein Fehler auf, so wird die Abarbeitung unterbrochen und die Schleife

kann nicht vollständig ausgeführt werden. Der erste Teil kann lokale Variablen definieren und initialisieren. Diese Zählvariable ist dann außerhalb des Blocks nicht mehr gültig.[25] Es darf noch keine lokale Variable mit dem gleichen Namen geben.

- **Schleifentest/Schleifenbedingung**
  Der mittlere Teil, wie `i<=10`, wird vor dem Durchlaufen des Schleifenrumpfs – also vor jedem Schleifeneintritt – getestet. Ergibt der Ausdruck `false`, wird die Schleife nicht durchlaufen und beendet. Das Ergebnis muss, wie bei einer `while`-Schleife, vom Typ `boolean` sein.
- **Schleifen-Inkrement durch einen Fortschaltausdruck**
  Der letzte Teil, wie `i++`, wird immer am Ende jedes Schleifendurchlaufs, aber noch vor dem nächsten Schleifeneintritt ausgeführt. Das Ergebnis wird nicht weiter verwendet. Ergibt die Bedingung des Tests `true`, dann befindet sich beim nächsten Betreten des Rumpfs der veränderte Wert im Rumpf.

Betrachten wir das Beispiel, so ist die Auswertungsreihenfolge folgender Art:

1. Initialisiere `i` mit 1.
2. Teste, ob `i <= 10` gilt.
3. Ergibt sich `true`, dann führe den Block aus, sonst ist es das Ende der Schleife.
4. Erhöhe `i` um 1.
5. Gehe zu Schritt 2.

Eine Endlosschleife

Da alle drei Ausdrücke im Kopf der Schleife optional sind, können sie weggelassen werden und es ergibt sich eine Endlosschleife. Diese Schreibweise ist somit semantisch äquivalent mit `while(true)`.

```
for ( ; ; )
  ;
```

Die trennenden Semikolons dürfen nicht verschwinden. Falls demnach keine Schleifenbedingung angegeben ist, ist der Ausdruck immer wahr. Es folgt keine Initialisierung und keine Auswertung des Fortschaltausdrucks.

Wann for und wann while?

Da sich `while`- und `for`-Schleife sehr ähnlich sind, besteht die berechtigte Frage, wann die eine und wann die andere zu nutzen ist. Leider verführt die kompakte `for`-Schleife sehr schnell zu einer Überladung. Manche Programmierer packen gerne alles in den Schleifenkopf hinein, und der Rumpf besteht nur aus einer leeren Anweisung. Dies ist ein schlechter Stil und muss vermieden werden. Die `for`-Schleife sollte dort eingesetzt werden, wo

25 Im Gegensatz zu C++ ist das Verhalten klar definiert, und es gibt kein hin und her. In C++ implementierten Compilerbauer die Variante einmal so, dass die Variable nur im Block gilt, andere interpretierten die Sprachspezifikation so, dass diese auch außerhalb gültig blieb. Die aktuelle C++-Definition schreibt nun vor, dass die Variable außerhalb des Blockes nicht mehr gültig ist. Da es jedoch noch alten Programmcode gibt, haben viele Compilerbauer eine Option eingebaut, mit der das Verhalten der lokalen Variablen bestimmt werden kann.

sich alle drei Ausdrücke im Schleifenkopf auf dieselbe Variable beziehen, etwa zum Durchzählen einer Variablen (*Zählvariable* oder *Laufvariable*):

```
// Zahlen von 1 bis 10 ausgeben

for ( int i = 1; i <= 10; i++ )
  System.out.println( i );
```

Vermieden werden sollten unzusammenhängende Ausdrücke im Schleifenkopf.

Geschachtelte Schleifen

Schleifen, und das gilt insbesondere für for-Schleifen, können geschachtelt werden. Syntaktisch ist das auch logisch, da sich innerhalb des Schleifenrumpfs beliebige Anweisungen aufhalten dürfen.

Beispiel Gib fünf Zeilen von Sternchen aus, wobei in jeder Zeile immer ein Stern mehr erscheinen soll. Als besonderes Element ist die Abhängigkeit des Schleifenzählers `j` von `i` zu werten.

```
for( int i = 1; i <= 5; i++ )
{
  for ( int j = 1; j <= i; j++ )
    System.out.print( "*" );

  System.out.println();
}
```

Es folgt die Ausgabe:

```
*
**
***
****
*****
```

Die übergeordnete Schleife nennt sich *äußere Schleife*, die untergeordnete *innere Schleife*. In unserem Beispiel wird die äußere Schleife die Zeilen zählen und die innere die Sternchen in eine Zeile ausgeben, also für die Spalte verantwortlich sein.

Da Schleifen beliebig tief geschachtelt werden können, muss besonders ein Auge auf die Laufzeit geworfen werden. Die inneren Schleifen werden immer so oft ausgeführt, wie die äußere Schleife durchlaufen wird.

for-Schleifen und ihr Komma-Operator

Im ersten und letzten Teil einer `for`-Schleife lässt sich ein Komma einsetzen. Damit lassen sich entweder mehrere Variablen gleichen Typs deklarieren – wie wir es schon kennen – oder mehrere Ausdrücke nebeneinander schreiben.[26]

26 Wenn Java eine ausdruckorientierte Sprache wäre, dann könnten wir hier beliebige Programme hineinlegen.

Beispiel Schleife mit zwei Zählern:

```
for ( int i = 1, j = 9; i <= j; i++, j-- )
  System.out.println( i + "*" + j + " = " + i*j );
```

Dann ist die Ausgabe:

```
1*9 = 9
2*8 = 16
3*7 = 21
4*6 = 24
5*5 = 25
```

Beispiel Berechne vor dem Schleifendurchlauf den Startwert für die Variablen `x` und `y`. Erhöhe dann `x` und `y` und führe die Schleife aus, bis `x` und `y` beide 10 sind.

```
int x, y;
for ( x = initX(), y = initY(), x++, y++;
      x == 10 && y == 10;
      x += xinc(), y += yinc() )
{
  // ...
}
```

Tipp Komplizierte `for`-Schleifen sind lesbarer, wenn die drei `for`-Teile in getrennten Zeilen stehen.

Wird das Komma für die Deklaration mehrerer Variablen verwendet, so kann dahinter kein Ausdruck mit Komma abgetrennt werden. Wenn der Compiler mit einer Deklaration beginnt, könnte er gar nicht zwischen einer zweiten Deklaration für eine Variable und dem folgenden Ausdruck unterscheiden, da das Komma die Variablennamen abtrennt.

Beispiel Folgende `for`-Schleife ergibt einen Fehler:

```
int i;
for ( int cnt = 0, i = 1; cnt < 10; cnt++ )
  ;
```

Der erste Teil leitet eine Anweisung ein. Nach dem Komma folgt für den Compiler aber die Deklaration einer zweiten Variablen. Da sie jedoch schon eine Zeile vorher definiert wurde, meldet der Compiler einen Fehler.

Auch umgekehrt funktioniert das nicht, denn eine Variablendeklaration ist kein Ausdruck, sie ist formal betrachtet eine Anweisung.

Beispiel Einer Anweisung kann keine Variablendeklaration folgen. Daher ist auch Folgendes falsch:

```
for ( i = 0, int j = 0; ; )
  ;
```

Im letzten Teil von `for`, dem Fortschaltausdruck, darf keine Variablendeklaration stehen. Wozu sollte das auch gut sein?

Zu den weiteren Einschränkungen gehört, dass es nicht möglich ist, Variablen unterschiedlicher Typen zu deklarieren, wie es etwa `for (int i=0, double d=0.0;;)` zeigt. Hier muss eine Variable außerhalb der `for`-Schleife definiert werden. Das Deklarieren einer äußeren Variable bringt vielleicht aber auch den unerwünschten Effekt mit sich, dass eine Variable, die eigentlich nur in der Schleife gültig sein soll, eine zu große Sichtbarkeit bekommt. Dann lässt sich ein Block aufspannen, in dem dann nur eine Variable gültig ist.

Beispiel Deklariere ein `double` und eine Ganzzahl-Variable für die Schleife.

```
{
  double d = 12;
  for ( int i = 0; i < 20; i++ )
    ;
}
// jetzt sind i und d wieder frei
double d = 12;
```

2.7.5 Ausbruch planen mit break und Wiedereinstieg mit continue

Wird innerhalb einer `for`-, `while`- oder `do`/`while`-Schleife eine `break`-Anweisung eingesetzt, so wird der Schleifendurchlauf beendet.

Beispiel Führe die Schleife so lange durch, bis `i` den Wert 0 hat.

```
int i = 10;

while ( true )
  if ( i-- == 0 )
    break;
```

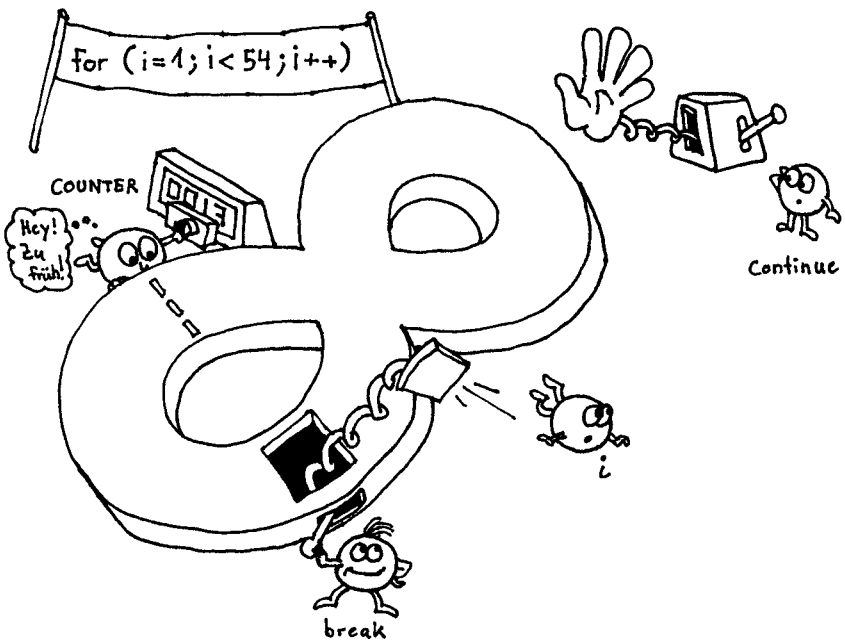
Die Anweisung ist nützlich, um im Programmblock festzustellen, ob die Schleife noch einmal durchlaufen werden soll. Sie entlastet den Schleifenkopf, der sonst die Bedingung testen würde. Da ein kleines `break` jedoch im Programmtext verschwinden könnte, die Bedeutung aber groß ist, sollte ein kleiner Hinweis auf diese Anweisung gesetzt werden.

Innerhalb einer `for`-, `while`- oder `do`/`while`-Schleife lässt sich eine `continue`-Anweisung einsetzen, die nicht wie `break` die Schleife beendet, sondern zum Schleifenkopf zurückgeht, so dass dort eine neue Prüfung durchgeführt werden kann, ob die Schleife weiter durchlaufen

werden soll. Ein häufiges Einsatzfeld sind Schleifen, die im Rumpf immer wieder Werte so lange holen und testen, bis sie passend zur Weiterverarbeitung sind.

Beispiel Gebe die geraden Zahlen von 0 bis 10 aus:

```
for ( int i = 0; i <= 10; i++ )
{
  if ( i%2 == 1 )
    continue;

  System.out.println( i + " ist eine gerade Zahl" );
}
```



2.7.6 break und continue mit Sprungmarken

Obwohl das Schlüsselwort `goto` in der Liste der reservierten Wörter auftaucht, ist diese Operation nicht erlaubt. Programmieren mit `goto` sollte vermieden werden. Mit dem Konzept von `break` lässt sich gut leben, und es kann auch noch ruhigen Gewissens eingesetzt werden. Doch zum Schrecken vieler kann `break` noch schmutziger eingesetzt werden, nämlich mit einer Sprungmarke. Das bringt Java verdächtig nahe in die `goto`-Welt der unstrukturierten Programmiersprachen, was die Entwickler eigentlich vermeiden wollten. Da jedoch Abbruchbedingungen – der häufigste Einsatzort eines `goto` – vereinzelt auftreten, wurden in Java `break` und `continue` mit Sprungmarken eingeführt.

Beispiel Der Einsatz von `break` oder `continue`:

```
one:
  while ( condition )
  {
    ...

  two:
    while ( condition )
    {
    ...
    // break oder continue
    }
  // nach two
  }
// nach one
```

Wird innerhalb der zweiten `while`-Schleife ein `break` platziert, dann würde es beim Aufruf die `while`-Schleife beenden. Das `continue` würde zur Fortführung der `while`-Schleife führen. Dieses Verhalten entspräche C, aber in Java ist es erlaubt, hinter den Schlüsselworten `break` und `continue` Sprungmarken zu setzen. Das C-Verhalten kann in Java mit `break two` oder `continue two` beschrieben werden. Dass aber auch beispielsweise `break one` möglich ist, zeigt die Mächtigkeit dieses Befehls. Durch `break` und `continue` mit Marken ist daher ein `goto` nahezu überflüssig.

2.8 Methoden einer Klasse

In objektorientierten Programmen interagieren zur Laufzeit Objekte miteinander und senden sich gegenseitig Nachrichten als Aufforderung, etwas zu machen. Diese Aufforderungen resultieren in einem Methodenaufruf, in dem Anweisungen stehen, die dann ausgeführt werden. Das Angebot eines Objekts, das, was es »kann«, wird in Java durch Methoden ausgedrückt. Die Begriffe »Methode« und »Funktion« wollen wir in diesem Tutorial gleichwertig benutzen.

Wir haben schon mindestens eine Methode kennen gelernt: `println()`. Sie ist eine Methode vom `out`-Objekt. Ein anderes Programmstück schickt nun eine Nachricht an das `out`-Objekt, die `println()`-Methode auszuführen. Im Folgenden werden wir den aktiven Teil des Nachrichtenversendens nicht mehr so genau betrachten, sondern wir sagen nur noch, dass eine Methode aufgerufen wird.

Die Operationen einer Klasse, also das Angebot eines Objekts, sind ein Grund für Funktionsdeklarationen in einer objektorientierten Programmiersprache. Daneben gibt es aber noch weitere Gründe, die für Methoden sprechen:

- Komplexe Programme werden in kleine Teilprogramme zerlegt, damit die Komplexität des Programms heruntergebrochen wird. Damit ist der Kontrollfluss leichter zu erkennen. In klassischen Programmen heißen die Methoden daher auch Unterprogramme. Dieses Wort wollen wir hier allerdings nicht benutzen.

- Wiederkehrende Programmteile sollen nicht immer wieder programmiert, sondern an einer Stelle angeboten werden. Änderungen an der Funktionalität lassen sich dann leichter durchführen, wenn der Code lokal zusammengefasst ist.

2.8.1 Bestandteil einer Funktion

Eine Funktion besteht aus mehreren Bestandteilen. Dazu gehören der *Methodenkopf* (kurz *Kopf*) und der *Methodenrumpf* (kurz *Rumpf*). Der Kopf besteht aus einem *Rückgabetyp* (auch *Ergebnistyp* genannt), dem *Funktionsnamen* und einer optionalen *Parameterliste*. Der Methodenname, die Parameter und die Typen der Parameter definieren die *Signatur* einer Methode. Pro Klasse darf es nur eine Methode mit derselben Signatur geben, sonst meldet der Compiler einen Fehler. Der Rückgabewert fließt nicht in die Signatur mit ein.

Beispiel Die Funktionen

```
Object habHunger( Object o )
```

und

```
void habHunger( Object p )
```

werden vom Compiler als gleich angesehen und können deshalb nicht zusammen in einer Klasse vorkommen.

Schauen wir uns noch zwei Beispiele aus der Java-API an.

Beispiel Betrachten wir eine Funktion, die es schon gibt und die in der API-Hilfe dokumentiert ist.

- `public static double max(double a, double b)`
- Returns the greater of two double values. That is, the result is the argument closer to positive infinity. If the arguments have the same value, the result is that same value. If either value is NaN, then the result is NaN. Unlike the numerical comparison operators, this method considers negative zero to be strictly smaller than positive zero. If one argument is positive zero and the other negative zero, the result is positive zero:

  Parameters:

  a – a double value.

  b – a double value.

  Returns:

  the larger of a and b.

Die Hilfe gibt Informationen über die komplette Signatur der Methode. Der Rückgabetyp ist ein `double`, die Funktion heißt `max()`, und sie erwartet genau zwei `double`-Zahlen. Verschwiegen haben wir die Schlüsselwörter `public static`, die so genannten Modifzierer. `public` gibt die Sichtbarkeit an und sagt, wer diese Funktion nutzen kann. Im Fall von `public` bedeutet es, dass jeder diese Funktion verwenden kann. Das Gegenteil ist `private`.

Dann kann nur das Objekt selbst diese Funktion nutzen. Das ist sinnvoll in dem Fall, wenn Funktionen benutzt werden, um die Komplexität zu verkleinern und Teilprobleme zu lösen. Private Funktionen werden in der Regel nicht in der Hilfe angezeigt. Das Schlüsselwort `static` zeigt an, dass sich die Funktion mit dem Klassennamen nutzen lässt, also kein Exemplar eines Objekts nötig ist.

Beispiel Es gibt Funktionen, die noch andere Modifizierer und eine erweiterte Signatur besitzen. Ein weiteres Beispiel aus der API:

▶ `protected final void implAccept(Socket s)`
`throws IOException`

Subclasses of ServerSocket use this method to override accept() to return their own subclass of socket. So a FooServerSocket will typically hand this method an empty FooSocket. On return from implAccept the FooSocket will be connected to a client:

Parameters:

s – the Socket

Throws:

IOException – if an I/O error occurs when waiting for a connection.

Since:

JDK1.1

Die Sichtbarkeit dieser Funktion ist `protected`. Das bedeutet, nur abgeleitete Klassen und Klassen im gleichen Verzeichnis (Paket) können diese Funktion nutzen. Ein zusätzlicher Modifizierer ist `final`, der in einer Vererbung der Unterklasse nicht erlaubt, die Funktion zu überschreiben und ihr neuen Programmcode zu geben. Zum Schluss folgt hinter dem Schlüsselwort `throw` eine Ausnahme. Dies sagt etwas darüber aus, welche Fehler die Funktion verursachen kann und worum sich der Programmierer kümmern muss. Im Zusammenhang mit der Vererbung werden wir noch über `protected` und `final` sprechen. Der Ausnahmebehandlung widmet sich ein eigenes Kapitel.

2.8.2 Aufruf

Da eine Funktion immer zu einer Klasse gehört, muss deutlich sein, zu wem die Methode gehört. Im Fall von `System.out.println()` ist `println()` eine Methode vom `out`-Objekt. Wenn wir das Maximum zweier Gleitkommazahlen mit `Math.max(a, b)` bilden, dann ist `max()` eine Funktion der Klasse `Math`. Für den Aufrufer ist damit immer ersichtlich, wer diese Methode anbietet, also auch, wer diese Nachricht entgegennimmt. Was der Aufrufer nicht sieht, ist die Arbeitsweise der Funktion. Der Funktionsaufruf verzweigt in den Programmcode, aber der Aufrufer weiß nicht, was dort geschieht. Er betrachtet nur das Ergebnis.

Die aufgerufene Funktion wird mit ihrem Namen genannt. Die Parameterliste wird durch ein Klammerpaar umschlossen. Diese Klammern müssen auch dann gesetzt werden, wenn die Methode keine Parameter enthält. Eine Funktion wie `System.out.println()` gibt nichts als Ergebnis einer Berechnung zurück. Anders ist die Funktion `max()`; sie liefert ein Ergebnis. Damit ergeben sich vier unterschiedliche Funktionentypen:

Funktion	Ohne Rückgabewert	Mit Rückgabewert
ohne Parameter	`System.out.println()`	`System.currentTimeMillis()`
mit Parameter	`System.out.println(4)`	`Math.max(12, 33)`

Tabelle 2.8 Funktionen mit Rückgabewerten und Parametern

Die Methode `System.currentTimeMillis()` gibt die Anzahl der verstrichenen Millisekunden ab dem 1.1.1970 als `long` zurück.

2.8.3 Methoden ohne Parameter

Die einfachste Funktion besitzt keinen Rückgabewert und keine Parameter. Im mathematischen Sinn ist dann vielleicht auch der Name »Funktion« falsch, wenn sie keinen Wert zurückliefert, aber das soll uns nicht kümmern. Der Funktionscode wird in geschweiften Klammern hinter den Kopf geschrieben und bildet damit den Körper der Methode. Gibt eine Funktion nichts zurück (das mathematische Dilemma), dann ist als spezieller Rückgabetyp `void` vorgesehen. Im klassischen Sinn ist dieser Typ von Funktion unter dem Namen »Prozedur« bekannt, die von der Aufgabe abstrahiert, indem sie Funktionalität hinter einem Namen verbirgt. Der Begriff »Prozedur« ist jedoch in der Objektwelt nicht anzutreffen.

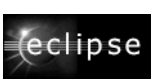
Beispiel Eine Funktion ohne Rückgabe und Parameter, die einfach etwas auf dem Bildschirm ausgibt

Listing 2.6 SimpleFunction.java

```
class SimpleFunction
{
  static void tollHier()
  {
    System.out.println( "Toll hier im Java-Land" );
  }
  public static void main( String args[] )
  {
    tollHier();
  }
```

Am Aufruf der Funktion lässt sich ablesen, dass hier kein Objekt gefordert ist, das mit der Methode verbunden werden soll. Das ist möglich, denn die Funktion ist als `static` deklariert, und innerhalb der Klasse lassen sich alle Funktionen einfach mit ihrem Namen nutzen.

Eine gedrückte Strg-Taste und ein Mausklick auf einen Bezeichner lässt Eclipse zur Deklaration springen. Ein Druck auf F3 hat den gleichen Effekt. Steht der Cursor in unserem Beispiel auf dem Methodenaufruf `tollHier()`, und F3 wird gedrückt, dann springt Eclipse zur Definition in Zeile 3 und hebt den Funktionsnamen vor.



2.8.4 Statische Methoden (Klassenmethoden)

Bisher arbeiten wir nur mit statischen Methoden (auch Klassenmethoden genannt), also mit Methoden, die ohne ein erzeugtes Objekt auskommen. Leicht fällt das Schlüsselwort `static` unter den Tisch, denn das `static` muss nicht zwingend vor der Methodendeklaration stehen – nur dann, wenn wir eine Methode aus einer anderen statischen Methode wie `main()` nutzen wollen. Fehlt das `static`, so erzeugt der Compiler etwa folgende Fehlermeldung: »non-static method hui() cannot be referenced from a static context«.

2.8.5 Parameter und Wertübergabe

Einer Funktion können Werte übergeben werden, die sie dann in ihre Arbeitsweise einbeziehen kann. Der Funktion `println(2001)` ist zum Beispiel ein Wert übergeben worden. Sie wird damit zur *parametrisierten Funktion*.

Beispiel Werfen wir einen Blick auf die Funktionsdefinition `max()` für Gleitkommazahlen. Die Methode soll später den größeren Wert auf dem Bildschirm ausgeben.

```
static void max( double a, double b )
{
  // Hier kommt die Implementierung hinein.
}
```

Der Bezeichner, der innerhalb der Methode verwendet wird, um den übergebenen Wert anzusprechen, heißt *formaler Parameter*. `a` und `b` sind in unserem Beispiel die formalen Parameter. Sie werden mit dem Komma getrennt aufgelistet. Für jeden Parameter muss ein Typ angegeben sein, und eine Abkürzung wie bei der Variablendeklaration `Typ V1,V2` ist nicht möglich. Jeder Parameter muss mit seinem eigenen Typ aufgelistet werden. Mehrere Bezeichner dürfen nicht den gleichen Namen tragen, andernfalls ergibt sich ein Übersetzungsfehler.

Der Aufrufer der Funktion gibt für jeden Parameter ein Argument an. Rufen wir unsere Methode `max()` etwa mit `max(10, y)` auf, so ist das Literal 10 und die Variable `y` ein Argument und ein *aktueller Parameter* der Funktion. Die Argumente müssen vom Typ her passen. Die Ganzzahl 10 kann auf ein `double` konvertiert werden, und `y` muss ebenfalls automatisch angepasst werden können. Für die Typanpassung gelten die bekannten Regeln.

Hinweis Anzahl der Parameter: Im Gegensatz zu C(++) muss beim Aufruf der Funktion die Anzahl der Parameter exakt stimmen. Eine variable Parameteranzahl – wie in C(++) durch »...« angedeutet – ist in Java nicht möglich. Alle Parameter sind fest und folglich typsicher.

Wertübergabe: Copy by Value

Wenn eine Funktion aufgerufen wird, dann gibt es in Java ein bestimmtes Verfahren, in dem jedes Argument einem Parameter übergeben wird. Diese Technik heißt im Allgemeinen *Parameterübergabemechanismus* und die meisten Programmiersprachen besitzen eine

ganze Reihe von verwirrenden Möglichkeiten. Java definiert nur einen Mechanismus, der *Wertübergabe* (engl. *copy by value*) genannt wird. Der in der Methode definierte Parameter wird als lokale Variable betrachtet, die zum Zeitpunkt des Aufrufs mit dem Argument initialisiert ist. Das Ende des Blocks bedeutet dann auch das Ende für die Parametervariable.

Beispiel Die Implementierung der Funktion `max()`:

```
static void max( double a, double b )
{
  if ( a > b )
    System.out.println( a );
  else
    System.out.println( b );
}
```

Innerhalb des Funktionskörpers nutzen wir einfach die übergebenen Werte über die Variable. Beim Aufruf werden die Werte des Arguments in die Variablen kopiert.

Der Wert von 10 gelangt in die Variable `a`, und der Inhalt von `i` wird ausgelesen und der Variablen `b` in der Methode zugänglich gemacht:

```
int i = 2;
max( 10, i );
```

Da der Wert der Variablen übergeben wird, heißt das insbesondere, dass es keine Übereinstimmung der Variablennamen geben muss. Die Variable `i` muss nicht `b` heißen. Wegen dieser Aufrufart kommt auch der Name »Copy by Value« zustande. Lediglich der Wert wird übergeben und kein Verweis auf die Variable, wie dies Referenzen in C++ tun.

Auswertung der Argumentenliste von links nach rechts

Bei einem Methodenaufruf werden erst alle Argumente ausgewertet und anschließend der Methode übergeben. Das bedeutet im Besonderen, dass Unterfunktionen ausgewertet und Zuweisungen gemacht werden können. Fehler führen dann zu einem Abbruch des Funktionsaufrufs. Bis zum Fehler werden alle Ausdrücke ausgewertet.

2.8.6 Methoden vorzeitig mit return beenden

Läuft eine Methode bis zum Ende durch, dann ist die Methode damit beendet und es geht zurück zum Aufrufer. In Abhängigkeit einer Bedingung kann eine Methode jedoch vor dem Ende des Ablaufs mit einer `return`-Anweisung beendet werden. Das ist nützlich bei Methoden, die in Abhängigkeit von Parametern vorzeitig aussteigen wollen. Wir können uns vorstellen, dass vor dem Ende der Funktion automatisch ein verstecktes `return` steht.

Beispiel Eine Methode soll die Wurzel einer Zahl auf dem Bildschirm ausgeben. Bei Zahlen kleiner Null erscheint eine Meldung, und die Methode wird verlassen. Andernfalls wird die Wurzelberechnung gemacht:

```
static void sqrt( double d )
{
  if ( d < 0 )
  {
    System.out.println( "Keine Wurzel aus negativen Zahlen!" );
    return;
  }
  System.out.println( Math.sqrt( d ) );
}
```

Die Realisierung wäre auch mit einer `else`-Anweisung möglich gewesen.

Eigene Methoden können natürlich wie Standardfunktionen heißen, da sie zu unterschiedlichen Klassen gehören.

2.8.7 Nicht erreichbarer Quellcode bei Funktionen

Folgt direkt hinter einer `return`-Anweisung Quellcode, so ist dieser nicht erreichbar – im Sinne von nicht ausführbar. `return` beendet also immer die Methode und kehrt zum Aufrufer zurück. Folgt nach dem `return` noch Quelltext, meldet der Compiler einen Fehler. In manchen Fällen ist das jedoch gewollt. Soll etwa eine Methode in der Testphase nicht komplett durchlaufen, sondern in der Mitte beendet werden, so lässt sich Folgendes nicht schreiben:

```
void faul()
{
  int i = 1;
  return;
  i = 2;              // Fehler!
}
```

Reduzieren wir eine Anweisung bis auf das Nötigste, das Semikolon, so führt dies bisweilen zu amüsanten Ergebnissen:

```
void f() {
  ;
  return;;
}
```

In diesem Beispiel sind zwei Null-Anweisungen enthalten. Eine vor dem `return` und eine dahinter. Doch das zweite Semikolon hinter dem `return` ist illegal, da es nicht erreichbarer Code ist.

2.8.8 Rückgabewerte

Funktionen wie die `Math.max()` liefern in Abhängigkeit von den Parametern ein Ergebnis zurück. Für den Aufrufer ist die Implementierung egal, er abstrahiert und nutzt lediglich die Methode als einen Ausdruck. Damit Methoden wie echte Funktionen Rückgabewerte an den Aufrufer weitergeben können, müssen zwei Dinge gelten:

- Eine Methode muss mit einem Rückgabetyp ungleich `void` definiert werden.
- Sie muss eine `return`-Anweisung besitzen, die einen geeigneten Wert zurückgibt.

Die allgemeine Syntax ist

```
return Ausdruck;
```

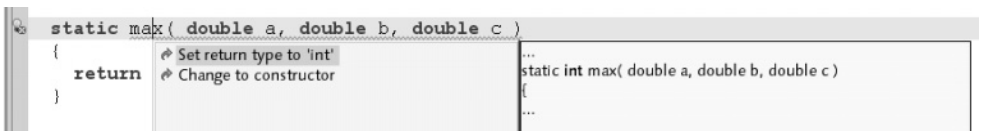
Ein allgemeines `return` ohne Ausdruck ist in einer Funktion mit Ergebnis ein Programmfehler. Der Rückgabewert muss an der Aufrufstelle jedoch nicht zwingend benutzt werden. Berechnet unsere Methode jedoch das Maximum zweier Zahlen, ist es unsinnig, den Rückgabewert nicht zu verwenden.

Hinweis Obwohl einige Programmierer den Ausdruck gerne klammern, ist das nicht nötig und soll auch nur komplexe Ausdrücke besser lesbar machen. Geklammerte Ausdrücke erinnern sonst nur an einen Funktionsaufruf, und diese Verwechslungsmöglichkeit sollte bei Rückgabewerten nicht bestehen.

Beispiel Eine Methode bildet aus drei Ganzzahlen das Maximum und gibt dieses zurück. Wir nutzen dazu eine Methode, die es schon gibt: `Math.max()`.

```
static double max( double a, double b, double c )
{
  return Math.max( Math.max( a, b), c );
}
```

Innerhalb der Funktion steckt wieder ein Funktionsaufruf. Der Typ hinter dem Rückgabewert muss kompatibel zum angegebenen Rückgabewert sein. Das passt in unserem Beispiel, denn die `Math.max()`-Funktion liefert ein `double`. Für die Anpassung gelten sonst wieder die bekannten Typanpassungsregeln.[27]



Eclipse erkennt, ob ein Rückgabetyp fehlt, und schlägt einen Typ vor. (Der in diesem Fall nicht passt!)

Beispiel Die nächste Funktion `isLeap()` stellt nach der Methode CMI fest, ob es sich bei einem Jahr um ein Schaltjahr handelt. Die Funktion arbeitet mit dem gregorianischen Kalender und gibt nur eine korrekte Antwort, wenn das Jahr zwischen 1583 und 20000 liegt.

27 Das dürfte relativ zukunftssicher sein.

Listing 2.7 LeapYear.java

```
class LeapYear
{

 /**
  * @return <code>true</code> wenn <code>year</code> ein Schaltjahr ist.
  * <code>false</code> sonst.
  */
  static boolean isLeap( int year )
  {
     return  year % 4 == 0
             && ( year % 100 != 0 || year % 400 == 0 );
  }

  public static void main( String args[] )
  {
    System.out.println( isLeap( 2000 ) );
  }
}
```

Mehrere return-Anweisungen

Für Methoden mit Rückgabewert gilt ebenso wie für `void`-Methoden, dass es mehr als ein `return` geben kann. Nach der Abarbeitung von `return` geht es im Programmcode des Aufrufers weiter wie bei den normalen `void`-Methoden.

Beispiel In `if`-Anweisungen mit weiteren `else-if`-Alternativen und Rücksprung ist die Semantik oft die Gleiche, wenn das `else-if` durch ein einfaches `if` ersetzt wird.

Der nachfolgende Programmcode zeigt das:

```
if ( a == 1 && b == 2 )
  return 0;
else if ( a == 2 && b == 1 )   // mit else
  return 1;
```

Dies lässt sich durch Folgendes ersetzen:

```
if ( a == 1 && b == 2 )
  return 0;

if ( a == 2 && b == 1 )        // ohne else
  return 1;
```

Passt die erste Bedingung, so endet die Funktion, und das nachfolgende `if` würde sowieso nicht vom Interpreter beachtet.

Wichtig ist nur, dass jeder denkbare Programmfluss mit einem `return` beendet wird. Der Compiler besitzt ein scharfes Auge und merkt, wenn es einen Programmpfad gibt, der nicht mit einem `return`-Ausdruck beendet wird.

Beispiel Die Funktion `gerade()` soll bei geraden Zahlen 1 und bei ungeraden Zahlen 0 liefern.

```
static int gerade( int i )
{
  switch ( i % 2 ) {
    case 0: return 1;
    case 1: return 0;
  }
}
```

Die Funktion lässt sich nicht compilieren, obwohl für uns der Rest der Division nur 0 oder 1 sein kann. Bei den Dingen, die für den Benutzer meistens offensichtlich sind, muss der Compiler passen, da er nicht hinter die Bedeutung sehen kann.

Ein weiteres Beispiel ist etwa eine Wochen-Funktion, die den Ganzzahl-Rückgabewert einer Funktion mit einem Wochentag als String kodiert verbindet. Wenn wir die Fälle 0=Sonntag bis 6=Samstag beachten, dann kann in unseren Augen ein Wochentag nicht 99 sein. Der Compiler kennt aber die Funktion nicht und weiß nicht, dass der Wertebereich beschränkt ist. Das Problem ließe sich mit einem `default` leicht beheben.

Beispiel Die Funktion `posOrNeg()` soll eine Zeichenkette mit der Information liefern, ob die übergebene Gleitkommazahl positiv oder negativ ist.

```
static String posOrNeg( double d )
{
  if ( d >= 0 )
    return "pos";
  if ( d < 0 )
    return "neg";
}
```

Es wird überraschen, aber dieser Programmcode ist ebenfalls fehlerhaft. Denn obwohl er offensichtlich für positive oder negative Zahlen den passenden String zurückgibt, gibt es einen Fall, den diese Funktion nicht abdeckt. Wieder gilt, dass der Compiler nicht erkennen kann, dass der zweite Ausdruck eine Negation des ersten sein soll. Es gibt aber noch einen zweiten Grund, der damit zu tun hat, dass es in Java spezielle Werte gibt, die keine Zahlen sind. Denn die Zahl `d` kann auch eine NaN (Not-a-Number) aus einer negativen Wurzel sein. Diesen speziellen Wert überprüft `posOrNeg()` nicht. Als Lösung für den einfachen Fall ohne NaN reicht es, aus dem zweiten `if` einfach ein `else` zu machen.

Bei Methoden, die einen Fehlerwert wie -1 zurückliefern, ist es eine häufig aufzufindende Implementierung, dass am Ende immer automatisch der Fehlerwert zurückgeliefert und dann in der Mitte die Methode bei passendem Ende verlassen wird.

Wir wollen nun eine Methode programmieren, die testet, ob ein Wert zwischen zwei Grenzen liegt. Dazu gibt es zwei Lösungen, wobei die meisten Programmierer zur ersten Lösung neigen.

Nennen wir unsere untere Schranke `a` und die obere Schranke `b`. Dann soll die Funktion `between()` testen, ob `x` zwischen `a` und `b` liegt. Bei Funktionen dieser Art ist es immer sehr wichtig, darauf zu achten und zu dokumentieren, ob der Test auf echt kleiner (`<`) oder kleiner gleich (`<=`) gemacht werden soll. Wir wollen hier auch die Gleichheit betrachten.

Die erste Lösungsidee zeigt sich in einer mathematischen Gleichung. Wir möchten gerne `a <= x <= b` schreiben, doch dies ist in Java nicht erlaubt.[28] So müssen wir einen Und-Vergleich anstellen, der etwa so lautet: Ist `a<=x && x<=b` dann liefere `true` zurück.

Die zweite Methode zeigt, dass sich das Problem auch ohne Und-Vergleich durch das Ausschlussprinzip lösen lässt:

```
static boolean between( int x, int a, int b )
{
  if ( x < a )
    return false;

  if ( x <= b )
    return true;

  return false;
}
```

Mit geschachtelten Anfragen sieht das dann so aus:

```
static boolean between( int x, int a, int b )
{
  if ( a <= x )
    if ( x <= b )
      return true;

  return false;
}
```

2.8.9 Methoden überladen

Eine Funktion ist gekennzeichnet durch Rückgabewert, Name, Parameter und unter Umständen durch Ausnahmefehler, die sie auslösen kann. Java erlaubt es, den Namen der Funktion gleich zu lassen, aber andere Parameter einzusetzen. Eine *überladene Methode* ist eine Funktion mit dem gleichen Namen wie eine andere Funktion, aber einer davon verschiedenen Parameterliste. Das geht auf zwei Arten:

- Eine Funktion kann gleich lauten, aber für den Compiler unterscheidbare Typen annehmen.
- Eine Funktion kann eine unterschiedliche Anzahl von Parametern akzeptieren.

Anwendungen für den ersten Fall gibt es viele. Der Name einer Funktion soll ihre Aufgabe beschreiben, aber nicht die Typen der Parameter, mit denen sie arbeitet, extra erwähnen. Das ist bei anderen Sprachen üblich, jedoch nicht in Java. Zum Beispiel bei der in der Mathe-Klasse `Math` angebotenen Funktion `max()`. Sie ist definiert auf unterschiedlichen

28 ... im Gegensatz zur Programmiersprache Python.

Typen, zum Beispiel `int` und `double`. Das ist viel schöner als die separaten Funktionen `maxInt()` und `maxDouble()` zu benutzen.

Beispiel Eine unterschiedliche Anzahl von Parametern ist ebenfalls eine sinnvolle Angelegenheit. Die Funktion `max()` könnten wir so für drei Parameter definieren.

Schreiben wir unsere eigene Variante für zwei und drei Parameter:

```
static int max( int i, int j ) {
  return Math.max( i, j );
}

static int max( int i, int j, int k ) {
  return max( i, max(j, k) );              // Methode von oben aufrufen.
}
```

Variable Parameterlisten wie in C(++) werden durch die Möglichkeit der überladenen Methoden nahezu unnötig.

Kommen wir nun zu zwei weiteren Beispielen, die etwas komplizierter sind und übersprungen werden können.

print() und println() sind überladen

Das bekannte `print()` ist eine überladene Funktion, die etwa wie folgt definiert ist:

```
class PrintStream
{
  void print( Object arg ) { ... }
  void print( String arg ) { ... }
  void print( char arg[] ) { ... }
}
```

Wird nun die Funktion `print()` mit irgendeinem Typ aufgerufen, dann wird die am besten passende Funktion herausgesucht. Versucht der Programmierer beispielsweise die Ausgabe eines Objekts `Date`, dann stellt sich die Frage, welche Methode sich darum kümmert. Glücklicherweise ist die Antwort nicht schwierig, denn es existiert auf jeden Fall eine `print()`-Methode, welche Objekte ausgibt. Und da `Date`, wie auch alle anderen Klassen, eine Unterklasse von `Object` ist, wird diese `print()`-Funktion gewählt. Natürlich kann nicht erwartet werden, dass das Datum in einem unbestimmten Format (etwa nur das Jahr) ausgegeben wird, jedoch wird eine Ausgabe auf dem Schirm sichtbar. Denn jedes Objekt kann sich durch den Namen identifizieren, und dieser würde in dem Fall ausgegeben. Obwohl es sich so anhört, als ob immer die Funktion mit dem Parameter-Objekt aufgerufen wird, wenn der Datentyp nicht angepasst werden kann, ist dies nicht ganz richtig. Wenn der Compiler keine passende Klasse findet, dann wird die nächste Oberklasse im Ableitungsbaum gesucht, für die in unserem Fall eine Ausgabefunktion existiert.

Negative Beispiele und schlaue Leute

Oft verfolgt auch die Java-Bibliothek die Strategie mit gleichen Namen und unterschiedlichen Typen. Es gibt allerdings ein paar Ausnahmen. In der Grafik-Bibliothek finden sich die Funktionen

- `drawString( String str, int x, int y )`,
- `drawChars( char data[], int offset, int length, int x, int y )` und
- `drawBytes( byte data[], int offset, int length, int x, int y )`.

Das ist äußerst hässlich und schlechter Stil.

Ein anderes Beispiel findet sich in der Klasse `DataOutputStream`. Hier heißen die Methoden etwa `writeInt()`, `writeChar()` und so weiter. Obwohl wir dies auf den ersten Blick verteufeln würden, ist diese Namensgebung sinnvoll. Ein Objekt vom Typ `DataOutputStream` dient zum Schreiben von primitiven Werten in einen Datenstrom, und davon gibt es bekanntermaßen einige, mit unterschiedlichen Längen. Gäbe es in `DataOutputStream` etwa eine Methode `write(int)` und `write(short)`, und wir fütterten sie mit `write(21)`, dann hätten wir das Problem, dass eine Typkonvertierung die Daten automatisch anpassen und der Datenstrom mehr Daten beinhalten würde als wir wünschen. Denn `write(21)` ruft etwa nicht `write(short)` auf und schreibt zwei Bytes, sondern es ruft `write(int)` auf und schreibt somit vier Bytes. Um also Übersicht über die geschriebenen Bytes zu behalten, ist eine ausdrückliche Kennzeichnung der Datentypen in manchen Fällen gar nicht so dumm.

2.8.10 Vorinitialisierte Parameter bei Funktionen

Überladene Funktionen lassen sich auch in dem Fall verwenden, wenn vorinitialisierte Werte bei nicht vorhandenen Parametern genutzt werden sollten. Ein Beispiel: Wir möchten eine Funktion zum Konvertieren von Zahlen kodiert als Zeichenkette in ein Zahlenformat erlauben. Dazu implementieren wir `toDecimal(String s, int radix)`. Wir möchten, dass `radix` automatisch 10 ist, wenn keine Basis angegeben ist. Die Sprache C++ erlaubt dies, Java jedoch nicht. Doch in Java überladen wir einfach die Funktion und rufen die andere Funktion mit 10 auf:

```
int toDecimal( String s, int radix )
{
  // Umwandlung
}
int toDecimal( String s )
{
  toDecimal( s, 10 );
}
```

2.8.11 Finale lokale Variablen

In einer Methode können Parameter oder lokale Variablen mit dem Modifizierer `final` deklariert werden. Dieses zusätzliche Schlüsselwort verbietet nochmalige Zuweisungen an diese Variable, so dass diese nicht mehr verändert werden kann. Dies gibt dem Compiler die Chance, zusätzliche Optimierungen vorzunehmen:

```
int foo( final int a )
{
    int i = 2;
    final int j = 3;

    i = 3;
//  j = 4;       führt zu einem Fehler
//  a = 2;       führt zu einem Fehler
}
```

Aufgeschobene Initialisierung

Java definiert die so genannte aufgeschobene Initialisierung. Das heißt im Zusammenhang mit finalen Werten, dass nicht zwingend zum Zeitpunkt der Variablendeklaration ein Wert zugewiesen werden muss. Dies kann auch genau einmal im Programmcode geschehen. Folgendes ist gültig:

```
final int a;
...
a = 2;
```

In der Vergangenheit enthielten Java- und Jikes-Compiler Fehler, so dass mehrfache Zuweisungen fälschlicherweise erlaubt waren.[29]

Obwohl auch Objektvariablen und Klassenvariablen final sein können, gibt es dort nur beschränkt eine aufgeschobene Initialisierung. Bei der Deklaration müssen wir die Variablen entweder direkt belegen oder im Konstruktor zuweisen. Wir werden uns dies später noch einmal genauer ansehen. Werden finale Variablen vererbt, so können Unterklassen diesen Wert auch nicht mehr überschreiben. (Das wäre ein Problem, aber vielleicht auch ein Vorteil für manche Konstanten.)

final in der Vererbung

In der Vererbung spielt das `final` bei Parametern keine Rolle. Wir können es als zusätzliche Information für die jeweilige Methode sehen. Eine Unterklasse kann demnach beliebig das `final` hinzufügen oder auch wegnehmen. Alte Bibliotheken lassen sich so leicht weiterverwenden.

2.8.12 Finale Referenzen in Objekten und das fehlende const

Wir haben gesehen, dass finale Variablen dem Programmierer vorgeben, dass er Variablen nicht beschreiben darf. Das heißt, Zuweisungen sind tabu. Leider löst das noch nicht das Problem, dass eine Methode mit übergebenen Referenzen Objektveränderungen vornehmen kann. Greifen wir etwas vor:

29 Ein Beispiel, das den Fehler reproduziert, findet der Leser unter *http://java-tutor.com/faq.html.*

Beispiel Die `foo()`-Methode ändert ein Attribut von einem `Point`-Objekt.

```
public void foo( final Point p )
{
//  p = new Point();
  p.x = 2;
}
```

Die Zuweisung ist für eine Referenz nicht weiter tragisch, da wir den Objektzustand dadurch nicht verändern, sondern lediglich die lokale Variable auf ein neues Objekt lenken. Nur Zuweisungen lässt `final` nicht zu. Was `final` nicht überprüft, ist, dass wir die Referenz auf der linken Seite einer Zuweisung haben können und dadurch in der Lage sind, das Objekt verändern zu können, wie im oberen Beispiel. `final` erfüllt demnach nicht die gewünschte Aufgabe, schreibende Objektzugriffe zu verhindern. Die Dokumentation muss also immer ausdrücklich beschreiben, wann die Funktion den Zustand eines Objekts modifiziert.

Obwohl die Java-Entwickler das Schlüsselwort `const` reserviert haben, ist diese Funktionalität noch nicht in die Sprache eingeflossen. Schade eigentlich. Wir sollten jedoch bemerken, dass es höchstens Schreibzugriffe anmerken könnte, Änderungen aber oft über `setXXX()`-Methoden realisiert werden.

2.8.13 Rekursive Funktionen

Wir wollen den Einstieg in die Rekursion mit einem kurzen Beispiel beginnen.

Auf dem Weg durch den Wald begegnet uns eine Fee. Sie spricht zu uns: »Du hast drei Wünsche frei«. Tolle Situation. Um das ganze Unglück aus der Welt zu räumen, entscheiden wir uns nicht für eine egozentrische Wunscherfüllung, sondern für die sozialistische. »Ich möchte Frieden für alle, Gesundheit und Wohlstand für jeden.« Und schwupps, so war es geschehen, und alle lebten glücklich bis ...

Einige Leser werden vielleicht die Hand vor den Kopf schlagen und sagen: »Quatsch! Ein Haus, ein Auto und einen Lebenspartner, der die Trägheit des Morgens duldet«. Glücklicherweise können wir das Dilemma mit der Rekursion lösen. Die Idee ist einfach – und in unseren Träumen schon erprobt – den letzten Wunsch als »Nochmal drei Wünsche frei« zu formulieren.

Beispiel Eine kleine Wunsch-Funktion:

```
static void fee()
{
  wunsch();
  wunsch();
  fee();
}
```

Durch den dauernden Aufruf der `fee()`-Funktion haben wir unendlich viele Wünsche frei. *Rekursion* ist also das Aufrufen der eigenen Methode, in der wir uns befinden. Dies kann auch über einen Umweg funktionieren. Das nennt sich dann nicht mehr *direkte Rekursion*, sondern *indirekte Rekursion*. Sie ist ein sehr alltägliches Phänomen, das wir auch von der Rückkopplung Mikrofon/Lautsprecher oder dem Blick mit einem Spiegel in den Spiegel kennen.

Abbruch der Rekursion

Wir müssen nun die Fantasie-Programme (deren Laufzeit und Speicherbedarf auch sehr schwer zu berechnen sind) gegen Java-Funktionen austauschen.

Beispiel Eine Endlos-Rekursion:

```
static void runter( int n )
{
  System.out.print( n + ", " );

  runter( n - 1 );
}
```

Rufen wir `runter(10)` auf, dann wird die Zahl 10 auf dem Bildschirm ausgegeben und anschließend `runter(9)` aufgerufen. Führen wir das Beispiel fort, so ergibt sich eine endlose Ausgabe, die so beginnt:

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, ...
```

An dieser Stelle erkennen wir, dass Rekursion prinzipiell etwas Unendliches ist. Für Programme ist dies aber ungünstig, wir müssen daher ähnlich wie bei Schleifen eine Abbruchbedingung formulieren und dann keinen Rekursionsaufruf mehr starten.

Beispiel Die Abbruchbedingung einer Rekursion:

```
static void runter( int n )
{
  if ( n == 0 )              // Rekursionsende
    return;

  System.out.print( n + ", " );

  runter( n - 1 );
}
```

Die `runter()`-Methode ruft jetzt nur noch so lange `runter(n-1)` auf, wie das `n` ungleich Null ist.

Unterschiedliche Rekursionsformen

Kennzeichen der bisherigen Programme war, dass nach dem Aufruf der Rekursion keine Anweisung stand, sondern die Methode mit dem Aufruf beendet wurde. Diese Rekursionsform nennt sich Endrekursion. Diese Form ist verhältnismäßig einfach zu verstehen. Schwieriger sind Rekursionen, bei denen hinter dem Methodenaufruf Anweisungen stehen. Betrachten wir folgende Methoden, in der die erste bekannt und die zweite neu ist.

```
static void runter1( int n )
{
  if ( n == 0 )   // Rekursionsende
    return;

  System.out.print( n + ", " );

  runter1( n - 1 );
}

static void runter2( int n )
{
  if ( n == 0 )   // Rekursionsende
    return;

  runter2( n - 1 );

  System.out.print( n + ", " );
}
```

Der Unterschied besteht darin, dass `runter1()` zuerst die Zahl `n` ausgibt und anschließend rekursiv `runter1()` aufruft. Die Methode `runter2()` steigt jedoch erst immer tiefer ab, und die Rekursion muss beendet sein, bis es zum ersten `print()` kommt. Daher gibt im Gegensatz zu `runter1()` die Methode `runter2()` die Zahlen in aufsteigender Reihenfolge aus:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

Dies ist einleuchtend, wenn wir die Ablaufreihenfolge betrachten. Beim Aufruf `runter2(10)` ist der Vergleich von `n` mit Null falsch, also wird ohne Ausgabe wieder `runter2(9)` aufgerufen. Ohne Ausgabe deshalb, da `print()` ja erst nach dem Funktionsaufruf steht. Es geht rekursiv tiefer bis `n` gleich Null ist. Dann beendet die letzte Methode mit `return`, und die Ausgabe wird nach dem `runter2()`, dem Aufrufer, fortgeführt. Dort ist `print()` die nächste Anweisung. Da wir nun noch tief verschachtelt stecken, gibt `print(n)` die Zahl 1 aus. Dann ist die Methode `runter2()` wieder beendet (ein unsichtbares, nicht direkt geschriebenes `return`), und sie springt zum Aufrufer zurück. Das war wieder die Methode `runter2()`, aber mit der Belegung `n=2`. Das geht soweit, bis es zurück zum Aufrufer kommt, der `runter(10)` aufgerufen hat, zum Beispiel die `main()`-Methode. Der Trick bei der Sache ist nun darin zu sehen, dass jede Methode ihre eigene lokale Variable besitzt.

Ausblick

Der niederländische Maler Maurits Cornelis Escher (1898–1972) machte die Rekursion auch in Bildern berühmt. Seiten mit Bildern und Vita finden sich zum Beispiel unter folgenden Webadressen:

- *http://www.worldofescher.com/*
- *http://www.etropolis.com/escher/*
- *http://www.iproject.com/escher/escher100.html*

Zwei weitere klassische Beispiele für Rekursionen sollen nachfolgend diskutiert werden.

2.8.14 Die Ackermann-Funktion

Wir wollen als mathematisch orientiertes Beispiel für eine rekursive Funktion die Ackermann-Funktion kennen lernen.[30] Sie ist nach F. Wilhelm Ackermann (1886–1962) benannt. Viele Funktionen der mathematischen Praxis sind primitiv rekursiv[31], und David Hilbert stellte 1926 die Frage, ob alle Funktionen, deren Argumente und Werte natürliche Zahlen sind, primitiv rekursiv sind. Die Ackermann-Funktion steigt sehr stark an und ist für Theoretiker ein Beispiel dafür, dass es berechenbare Funktionen gibt, die aber nicht primitiv rekursiv sind. Im Jahre 1928 zeigte Ackermann dies an einem Beispiel: der Ackermann-Funktion.[32] Sie wächst stärker als es Substitution und Rekursion ermöglichen und nur für kleine Argumente lassen sich die Funktionswerte noch ohne Rekursion berechnen. Darin bestand auch die Beweisidee von Ackermann, eine Funktion zu definieren, die schneller wächst als alle primitiv rekursiven Funktionen. Wir wollen hier nicht die originale Version von Ackermann benutzen, die durch die Funktionalgleichung

$$f(n', x', y') = f(n, f(n', x, y), x)$$

ausgedrückt wird, sondern die vereinfachte Variante von Hans Hermes. Wir wollen die Version von Hermes aber fortan auch Ackermann-Funktion nennen, da sie direkt aus dem Original gewonnen wird. Für die oben angegebene Funktion muss in der Abhandlung von Ackermann nachgeblättert werden, um den Nachweis des Nicht-primitiv-rekursiv-Seins zu finden.

Die neue Ackermann-Funktion ist eine Abbildung von zwei ganzen Zahlen auf eine ganze Zahl $a(n,m)$. Sie ist mathematisch durch folgende Gesetzmäßigkeit definiert:

$$a(0,m) = m + 1$$
$$a(n,0) = a(n - 1, 1)$$
$$a(n,m) = a(n - 1, a(n, m - 1)).$$

Die Ackermann-Funktion ist dafür berühmt, die Rechenkapazität ganz schnell zu erschöpfen. Sehen wir uns die Implementierung in Java an und testen wir das Programm mit ein paar Werten.

30 Das macht dann auch die Informatiker glücklich ...

31 Für Theoretiker: Eine Funktion heißt `primitiv-rekursiv`, falls sie elementar ist oder in endlich vielen Schritten aus einer elementaren Funktion durch Ersetzung (Substitution) und Rekursion hervorgeht.

32 Nachzulesen in »Zum Hilbertschen Aufbau der reellen Zahlen« von Ackermann, W. Math. Ann 99, 118-133 (1928).

Listing 2.8 Ackermann.java

```
class Ackermann
{
  public static long ackermann( long n, long m )
  {
    if ( n == 0 )
      return m + 1;
    else

      if ( m == 0 )
        return ackermann( n - 1, 1 );
      else
        return ackermann( n - 1, ackermann(n, m - 1) );

  }

  public static void main( String args[] )
  {
    int x = 2,
        y = 2;

    System.out.println( "ackermann(" + x + "," + y + ")=" + ackermann(x,y) );
  }
}
```

Für den Aufruf `ackermann(1, 2)` veranschaulicht die folgende Ausgabe die rekursive Eigenschaft der Ackermann-Funktion. Die Stufen der Rekursion sind durch Einrückungen deutlich gemacht:

```
a(1,2):
 a(0,a(1,1)):
  a(0,a(1,0)):
   a(1,0):
    a(0,1)=2
   a(1,0)=2
   a(0,2)=3
  a(0,a(1,0))=3
  a(0,3)=4
 a(0,a(1,1))=4
a(1,2)=4
```

Bei festen Zahlen lässt sich der Wert der Ackermann-Funktion direkt angeben.

$a(1,n) = n + 2$
$a(2,n) = 2n + 3$
$a(3,n) = 2^{n+3} - 3$

Für große Zahlen übersteigt die Funktion aber schnell alle Berechnungsmöglichkeiten.

2.8.15 Die Türme von Hanoi

Die Legende der Türme von Hanoi soll erstmalig von Ed Lucas in einem Artikel der französischen Zeitschrift »Cosmo« im Jahre 1890 veröffentlicht worden sein. Wir halten uns hier an eine Überlieferung von C.H. A. Koster aus dem Buch »Top-down Programming with Elan«, Ellis Horwood 1987.

Der Legende nach standen vor langer Zeit im Tempel von Hanoi drei Säulen. Die erste Säule war aus Kupfer, die zweite aus Silber und die dritte aus Gold. Auf der Kupfersäule waren einhundert Scheiben aufgestapelt. Die Scheiben hatten in der Mitte ein Loch und waren aus Porphyr[33]. Die Scheibe mit dem größten Umfang lag unten und alle kleiner werdenden Scheiben oben auf. Ein alter Mönch stellte sich die Aufgabe, den Turm der Scheiben von der Kupfersäule zur Goldsäule zu bewegen. In einem Schritt sollte aber nur eine Scheibe bewegt werden und zudem war die Bedingung, dass eine größere Scheibe niemals auf eine kleinere bewegt werden durfte. Der Mönch erkannte schnell, dass er die Silbersäule nutzen musste; er setzte sich an einen Tisch, machte einen Plan, überlegte und kam zu einer Entscheidung. Er konnte sein Problem in drei Schritten lösen.

Am nächsten Tag schlug der Mönch die Lösung an die Tempeltür:

- Falls der Turm aus mehr als einer Scheibe besteht, bitte Deinen ältesten Schüler, einen Turm von (n – 1) Scheiben von der ersten zur dritten Säule unter Verwendung der zweiten Säule umzusetzen.
- Trage selbst die erste Scheibe von einer zur anderen Säule.
- Falls der Turm aus mehr als einer Scheibe besteht, bitte Deinen ältesten Schüler, einen Turm aus (n – 1) Scheiben von der dritten zu der anderen Säule unter Verwendung der ersten Säule zu transportieren.

Und so rief der alte Mönch seinen ältesten Schüler zu sich und trug ihm auf, den Turm aus 99 Scheiben von der Kupfersäule zur Goldsäule unter Verwendung der Silbersäule umzuschichten und ihm den Vollzug zu melden.

Nach der Legende würde das Ende der Welt nahe sein, bis der Mönch seine Arbeit beendet hätte. Nun, soweit die Geschichte. Wollen wir den Algorithmus zur Umschichtung der Porphyrscheiben in Java programmieren, so machen wir dies sicherlich rekursiv.

Listing 2.9 Hanoi.java

```
class Hanoi
{
  static void bewegeScheibe( int n, String von, String nach )
  {
    System.out.println( "Scheibe " + n + " von " + von +
    " nach " + nach );
  }
  static void versetzeTurm( int n, String kupfer,
                            String silber, String gold )
  {
```

33 Gestein, das aus der porphyrischen Etschplattform gewonnen wird. Dies ist eine Gebirgsgruppe vulkanischen Ursprungs in Trentino/Südtirol. Besondere Eigenschaften von Porphyr sind: hohe Bruchfestigkeit, hohe Beständigkeit gegen physikalisch-chemische Wirkstoffe und hohe Wälz- und Gleitreibung.

```
    if ( n > 1 )
    {
      versetzeTurm( n-1, kupfer, gold, silber );
      bewegeScheibe( n, kupfer, gold );
      versetzeTurm( n-1, silber, kupfer, gold );
    }
    else
          bewegeScheibe( n, kupfer, gold );
      }
  public static void main( String args[] )
  {
    versetzeTurm( 4, "Kupfer", "Silber", "Gold" );
  }
}
```

Starten wir das Programm mit vier Scheiben, so bekommen wir folgende Ausgabe:

```
Scheibe 1 von Kupfer nach Silber
Scheibe 2 von Kupfer nach Gold
Scheibe 1 von Silber nach Gold
Scheibe 3 von Kupfer nach Silber
Scheibe 1 von Gold nach Kupfer
Scheibe 2 von Gold nach Silber
Scheibe 1 von Kupfer nach Silber
Scheibe 4 von Kupfer nach Gold
Scheibe 1 von Silber nach Gold
Scheibe 2 von Silber nach Kupfer
Scheibe 1 von Gold nach Kupfer
Scheibe 3 von Silber nach Gold

Scheibe 1 von Kupfer nach Silber
Scheibe 2 von Kupfer nach Gold
Scheibe 1 von Silber nach Gold
```

Schon bei vier Scheiben haben wir 15 Bewegungen. Nun wollen wir uns die Komplexität bei *n* Porphyrscheiben überlegen. Bei einer Scheibe haben wir nur eine Bewegung zu machen. Bei zwei Scheiben aber schon doppelt so viele wie vorher und noch eine zusätzlich. Formaler:

$S_1 = 1$
$S_2 = 1 + 2S_1 = 3$
$S_3 = 1 + 2S_2 = 7$

Führen wir die Berechnung induktiv fort, so folgt für *Sn*, dass $2n - 1$ Schritte auszuführen sind, um *n* Scheiben zu bewegen. Nehmen wir an, unser Prozessor arbeitet mit 100 MIPS, also 100 Millionen Operationen pro Sekunde, dann ergibt sich für $n = 100$ eine Zeit von $4*10^{13}$ Jahren (etwa 20.000 geologische Erdzeitalter). An diesem Beispiel wird uns wie beim Beispiel mit der Ackermann-Funktion deutlich: Die Funktionen sind im Prinzip berechenbar, nur praktisch ist so ein Algorithmus nicht.

2.9 Weitere Operatoren

2.9.1 Bitoperationen

Zahlen und Werte sind im Computer als Sammlung von Bits gegeben. Ein Bit ist ein Informationsträger für den Wert wahr oder falsch. Bits werden zusammengesetzt zu Folgen wie einem Byte, das aus acht Bits besteht. Die Belegung der Bits ergibt einen Wert. Wenn also jedes der acht Bits unterschiedliche Belegungen annehmen kann, so ergeben sich 256 unterschiedliche Zahlen. Jede Stelle im Bit bekommt dabei eine Wertigkeit zugeordnet.

Beispiel Die Wertebelegung für die Zahl 19. Sie ist zusammengesetzt aus einer Anzahl von Summanden der Form 2^n. Die Zahl 19 berechnet sich aus 16+2+1. Genau diese Bits sind gesetzt.

Bit	7	6	5	4	3	2	1	0
Wertigkeit	2^7=128	2^6=64	2^5=32	2^4=16	2^3=8	2^2=4	2^1=2	2^0=1
Belegung für 19	0	0	0	1	0	0	1	1

Tabelle 2.9 Bitbelegung für die Zahl 19

Mit Bitoperatoren lassen sich Binäroperationen auf Operanden durchführen, um beispielsweise Bits eines Worts zu setzen. Zu den Bitoperationen zählen Verknüpfungen, Schiebeoperationen und das Komplement. Durch die bitweisen Operatoren können die einzelnen Bits abgefragt und manipuliert werden. Als Verknüpfungen bietet Java die folgenden Bitoperatoren an:

Operator	Bezeichnung	Funktion
~	Komplement	Alle Bits werden invertiert.
|	bitweises Oder	Bei `a|b` wird jedes Bit von `a` und `b` einzeln Oder-verknüpft.
&	bitweises Und	Bei `a&b` wird jedes Bit von `a` und `b` einzeln Und-verknüpft.
^	bitweises exklusives Oder (Xor)	Bei `a^b` wird jedes Bit von `a` und `b` einzeln Xor-verknüpft.

Tabelle 2.10 Bitoperatoren in Java

Betrachten wir allgemein die binäre Verknüpfung `a # b`. Bei der binären bitweisen Und-Verknüpfung mit `&` gilt für jedes Bit: Nur wenn beide Operanden `a` und `b` 1 sind, dann ist auch das Ergebnis 1. Bei der Oder-Verknüpfung mit | muss nur einer der Operanden 1 sein, damit das Ergebnis 1 ist. Bei einem exklusiven Oder (Xor) ist das Ergebnis 1, wenn nur genau einer der Operanden 1 ist. Sind beide gemeinsam 0 oder 1, ist das Ergebnis 0. Dies entspricht einer binären Addition oder Subtraktion. Fassen wir das Ergebnis noch einmal in einer Tabelle zusammen:

Bit 1	Bit 2	~Bit 1	Bit 1 & Bit 2	Bit 1 | Bit 2	Bit 1 ^ Bit 2
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Tabelle 2.11 Bitoperatoren in einer Wahrheitstafel

2.9.2 Vorzeichenlose Bytes in ein Integer und Char konvertieren

Liegt ein Byte als Datentyp vor, so kann es zwar automatisch zu einem `int` angepasst werden, aber die automatische Typkonvertierung erfüllt nicht immer den gewünschten Zweck:

```
byte b1 =       100;
byte b2 = (byte)200;

System.out.println( b1 );                  // 100
System.out.println( b2 );                  // -56
```

Beim zweiten Byte ist eine Typanpassung nötig, da 0x90 größer als 0x7f ist und daher den Zahlenbereich eines Byte -128 bis 127 überschreitet. Dennoch kann ein Byte das gegebene Bitmuster annehmen und repräsentiert dann die negative Zahl -56. Wird diese ausgegeben, findet bei `println(int)` eine automatische Typanpassung auf ein `int` statt und die negative Zahl als `byte` wird zur negativen `int`-Zahl. Das bedeutet, dass das Vorzeichen übernommen wird. In einigen Fällen ist es jedoch wünschenswert, ein `byte` vorzeichenlos zu behandeln. Bei der Ausgabe soll dann ein Datenwert zwischen 0 und 255 herauskommen. Um das zu erreichen, schneiden wir mit der Und-Verknüpfung alle anderen Bits ausgenommen die unteren acht heraus. Das reicht schon. Die Lösung zeigt die Funktion `byteToInt()`:

```
static int byteToInt( byte b )
{
  return b & 0xff;
}
```

Die Schreibweise `(int)b` ist nicht nötig, da die automatische Typanpassung das Byte `b` in einer arithmetischen Operation automatisch auf ein `int` konvertiert.

Mit einer ähnlichen Arbeitsweise können wir auch die Frage lösen, wie sich ein Byte, dessen Integerwert im Minusbereich liegt, in ein `char` konvertieren lässt. Der erste Ansatz über eine Typumwandlung `(char)byte` ist falsch, und auf der Ausgabe dürfte nur ein rechteckiges Kästchen oder ein Fragezeichen erscheinen:

```
byte b = (byte) 'ß';
System.out.println( (char) b );    // Ausgabe ist  ?
```

Das Dilemma ist wieder die fehlerhafte Vorzeichenanpassung. Bei der Benutzung des Bytes wird es zuerst in ein `int` konvertiert. Das »ß« wird dann zu -33. Im nächsten Schritt wird diese -33 dann zu einem `char` umgesetzt. Das ergibt 65503, was einen Unicode-Bereich

trifft, der zur Zeit kein Zeichen definiert. Es wird wohl auch noch etwas dauern, bis die ersten Außerirdischen uns neue Zeichensätze schenken. Gelöst wird der Fall wie oben, in dem von `b` nur die unteren acht Bits betrachtet werden. Das geschieht wieder durch ein Ausblenden über den Und-Operator. Damit ergibt sich korrekt:

```
char c = (char) (b & 0x00ff);
```

2.9.3 Variablen mit Xor vertauschen

Eine besonders trickreiche Idee für das Vertauschen von Variableninhalten arbeitet mit der Xor-Funktion und benötigt keine temporäre Zwischenvariable. Die Zeilen zum Vertauschen von `x` und `y` lauten wie folgt:

```
int x = 12,
    y = 49;

x ^= y;  // x = x ^ y
y ^= x;  // y = y ^ x
x ^= y;  // x = x ^ y

System.out.println( x + " " + y);  // Ausgabe ist: 49 12
```

Der Trick funktioniert, da wir mit Xor etwas »rein- und rausrechnen« können. Zuerst rechnen wir in der ersten Zeile das y in das x. Wenn wir anschließend die Zuweisung an das y machen, dann ist das der letzte schreibende Zugriff auf `y`, also muss hier schon das vertauschte Ergebnis stehen. Das stimmt auch, denn expandieren wir die zweite Zeile, steht dort »y wird zugewiesen an y ^ x« und dies ist y ^ (x ^ y). Der letzte Ausdruck verkürzt sich zu y = x, da aus der Definition der Xor-Funktion für einen Wert a hervorgeht a ^ a = 0. Die Zuweisung hätten wir zwar gleich so schreiben können, aber dann wäre der Wert von y verloren gegangen. Der steckt aber noch in x aus der ersten Zuweisung. Betrachten wir daher die letzte Zeile x ^ y. y hat den Startwert von x, doch in x steckt ein Xor-y. Daher ergibt x ^ y den Wert x ^ x ^ y, und der verkürzt sich zu y. Demnach haben wir den Inhalt der Variablen vertauscht. Im Übrigen können wir für die drei Xor-Zeilen alternativ schreiben:

```
y ^= x ^= y;   // Auswertung automatisch y ^= (x ^= y)
x ^= y;
```

Da liegt es doch nahe, die Ausdrücke weiter abzukürzen zu x ^= y ^= x ^= y. Doch leider ist das falsch (es kommt für x immer Null heraus). Den motivierten Lesern bleibt dies als Denksportaufgabe.

2.9.4 Die Verschiebeoperatoren

Unter Java gibt es drei *Verschiebeoperatoren* (engl. *shift-operator*). Sie bewegen die Bits eines Datenworts in eine Richtung. Obwohl es nur zwei Richtungen rechts und links gibt, muss noch der Fall betrachtet werden, ob das Vorzeichen beim Links-Shift beachtet wird oder nicht.

i << n

Der Operand `i` wird unter Berücksichtigung des Vorzeichens *n*-mal nach links geschoben (mit 2 multipliziert). Der rechts frei werdende Bitplatz wird mit 0 aufgefüllt. Das Vorzeichen ändert sich jedoch, sobald eine 1 von der Position MSB – 1 nach MSB geschoben wird. MSB steht hier für *Most Significant Bit*, also das Bit mit der höchsten Wertigkeit in der binären Darstellung.

Das ist jedoch akzeptabel, da es ja ohnehin zu einem Bereichsüberlauf gekommen wäre.

i >> n

Da es in Java nur vorzeichenbehaftete Datentypen gibt, kommt es unter Berücksichtigung des Vorzeichens beim normalen Rechts-Shift zu einer vorzeichenrichtigen Ganzzahldivision durch 2. Dabei bleibt das linke Vorzeichen-Bit unberührt. Je nachdem, ob das Vorzeichen-Bit gesetzt ist oder nicht, wird eine 1 oder eine 0 von links eingeschoben.

Hinweis Die herausgeschobenen Bits sind für immer verloren!

```
System.out.println( 65535 >> 8 );  // 255
System.out.println( 255 << 8);     // 65280
```

Es ist 65535 = 0xFFFF und nach der Rechtsverschiebung 65535 >> 8 ergibt sich 0x00FF = 255. Schieben wir nun wieder nach links, also 0x00FF << 8, dann ist das Ergebnis 0xFF00 = 65280.

i >>> n

Der Operator >>> verschiebt eine Variable (Bitmuster) bitweise um *n* Schritte nach rechts, ohne das Vorzeichen der Variablen zu berücksichtigen (vorzeichenloser Rechts-Shift). So werden auf der linken Seite (MSB) nur Nullen eingeschoben; das Vorzeichen wird mitgeschoben. Bei einer positiven Zahl hat dies keinerlei Auswirkungen, und das Verhalten ist wie beim >>-Operator.

Beispiel Verschiebung einer positiven und negativen Zahl:

Listing 2.10 ShiftRightDemo.java

```
class ShiftRightDemo
{
  public static void main( String args[] )
  {
    int i = 64;
    int j = i >>> 1;

    System.out.println( " 64 >>> 1 = " + j );

    i = -64;
```

```
    j = i >>> 1;

    System.out.println( "-64 >>> 1 = " + j );
  }
}
```

Die Ausgabe ist für den negativen Operanden besonders spannend:

```
 64 >>> 1 = 32
-64 >>> 1 = 2147483616
```

Ein <<<-Operator macht allerdings keinen Sinn, da beim Links-Shiften sowieso nur Nullen eingefügt werden.

Division und Multiplikation mit Verschiebeoperatoren

Wird ein Wert um eine Stelle nach links geschoben, so kommt in das niedrigste Bit (das Bit ganz rechts) eine Null hinein und alle Bits schieben sich eine Stelle weiter. Das Resultat ist, dass die Zahl mit 2 multipliziert wird. Natürlich müssen wir mit einem Verlust von Informationen rechnen, wenn das höchste Bit gesetzt ist, denn dann wird es herausgeschoben, aber das Problem haben wir auch schon bei der normalen Multiplikation. Es gibt in Java keinen Operator, der die Bits rollt, der also die an einer Stelle herausfallenden wieder an der anderen Seite einfügt.

Wenn ein einmaliger Shift nach links mit 2 multipliziert, so würde eine Verschiebung um zwei Stellen nach links eine Multiplikation mit 4 bewirken. Allgemein gilt: Bei einem Shift von *i* nach links ergibt sich eine Multiplikation mit 2^i. Wir können dies dazu nutzen, beliebige Multiplikationen durch Verschiebung nachzubilden.

Beispiel Multiplikation mit 10 durch Verschiebung der Bits:

```
10*n == n<<3 + n<<1
```

Das funktioniert, da 10*n = (8+2)*n = 8*n + 2*n gilt.

Diese Umsetzung ist nicht immer einfach, und es gibt tatsächlich kein Verfahren, welches eine optimale Umsetzung liefert. Doch arbeiteten viele Prozessoren auf diese Weise intern die Multiplikation ab, und ein Compiler nutzt dies gern zur Optimierung der Laufzeit. Eine Verschiebeoperation ist bei vielen Prozessoren schneller als eine Multiplikation. Doch ist hier Obacht zu geben, denn eine lange Folge von Verschiebungen ist nicht schneller, sondern langsamer als eine direkte Multiplikation.

Neben der Addition kommt selbstverständlich auch die Subtraktion in Frage. Ersetzen wir im oberen Beispiel das Plus durch ein Minus, so bekämen wir eine Multiplikation mit 6. Natürlich müssen wir auf die Überläufe der Zwischenergebnisse bei großen Zahlen achten. Diese würde es bei einer echten Multiplikation nicht geben.

Was wir am Beispiel der Verschiebung nach links gezeigt haben, funktioniert genauso mit einem Shift nach rechts. Jetzt wird bei einmaliger Verschiebung durch 2 dividiert. Jetzt können beliebige Ausdrücke mit * und / und einer Konstante auf Verschiebungen und einfachen arithmetischen Operationen abgebildet werden.

Die Bitoperatoren in Assembler verglichen mit <<<, << und >>

Auch in Assembler gibt es zwei Gruppen von Schiebeoperatoren: die arithmetischen Schiebebefehle (SAL und SAR), die das Vorzeichen des Operanden beachten, und die logischen Schiebebefehle (SHL und SHR), die den Operanden ohne Beachtung eines etwaigen Vorzeichens schieben. Die Befehle SAL und SHL haben die gleiche Wirkung. So ist `>>>` der Bitoperator, in dem das Vorzeichen nicht beachtet wird, wie SHR in Assembler. Es gibt in Java auch keinen Bitoperator `<<<`, da – wie in Assembler – SAL = SHL gilt (`<<<` würde die gleiche Wirkung haben wie `<<`).

Bits rotieren

Java hat zwar Operatoren zum Verschieben von Bits, aber nicht zum Rotieren. Beim Rotieren werden Bits um eine bestimmte Stelle verschoben, die herausfallenden Bits kommen aber auf der anderen Seite wieder rein.

Eine Funktion ist leicht geschrieben: Der Trick dabei ist, die herausfallenden Bits vorher zu extrahieren und auf der anderen Seite wieder einzusetzen.

```
public static int rotateLeft( int v, int n )
{
  return (v << n) | (v >>> (32 - n));
}

public static int rotateRight( int v, int n )
{
  return (v >>> n) | (v << (32 - n));
}
```

Die Funktionen rotieren jeweils n Bits nach links oder rechts. Da der Datentyp `int` ist, ist die Verschiebung `n` in dem Wertebereich von 0 bis 31 erlaubt.

2.9.5 Setzen, Löschen, Umdrehen und Testen von Bits

Die Bitoperatoren lassen sich zusammen mit den Shift-Operatoren gut dazu verwenden, ein Bit zu setzen respektive herauszufinden, ob ein Bit gesetzt ist. Betrachten wir folgende Funktionen, die ein bestimmtes Bit setzen, abfragen, invertieren und löschen.

Beispiel Anwendung aller Bitoperatoren:

```
int setBit( int n, int pos )
{
  return n | (1 << pos);
}
```

```
int clearBit( int n, int pos )
{
  return n & ~ (1 << pos);
}

int flipBit( int n, int pos )
{
  return n ^ (1 << pos);
}
boolean testBit( int n, int pos )
{
  int mask = 1 << pos;

  return (n & mask) == mask;
  // alternativ: return (n & 1<<pos)!=0;

}
```

2.9.6 Der Bedingungsoperator

In Java gibt es ebenso wie in C(++) einen Operator, der drei Operanden benutzt. Dies ist der Bedingungsoperator, der auch Konditionaloperator, ternärer Operator beziehungsweise trinärer Operator genannt wird. Er erlaubt es, den Wert eines Ausdrucks von einer Bedingung abhängig zu machen, ohne dass dazu eine `if`-Anweisung verwendet werden muss. Die Operanden sind durch `?` beziehungsweise `:` voneinander getrennt:

```
ConditionalOrExpression ? Expression : ConditionalExpression
```

Der erste Ausdruck muss vom Typ `boolean` sein und bestimmt, ob das Ergebnis `Expression` oder `ConditionalExpression` ist. Der Bedingungsoperator kann eingesetzt werden, wenn der zweite und dritte Operand ein numerischer Typ, boolescher Typ oder Referenztyp ist. Der Aufruf von Methoden, die demnach `void` zurückgeben, ist nicht gestattet.

Eine Anwendung für den trinären Operator ist oft eine Zuweisung an eine Variable:

```
Variable = Bedingung ? Ausdruck1 : Ausdruck2;
```

Der Wert der Variablen wird jetzt in Abhängigkeit von der Bedingung gesetzt. Ist sie erfüllt, dann erhält die Variable den Wert des ersten Ausdrucks, andernfalls wird der Wert des zweiten Ausdrucks zugewiesen.

Beispiel So etwa für ein Maximum:

```
max = ( a > b ) ? a : b;
```

Dies entspricht beim herkömmlichen Einsatz für `if`/`else`:

```
if ( a > b )
  max = a;
else
  max = b;
```

Mit dem Rückgabewert können wir alles Mögliche machen, etwa ihn direkt ausgeben. Das wäre mit `if`/`else` nur mit temporären Variablen möglich.

```
System.out.println( ( a > b ) ? a : b );
```

Beispiele

Der Bedingungsoperator findet sich häufig in kleinen Funktionen. Dazu einige Beispiele:

Beispiel Um das Maximum oder Minimum zweier Ganzzahlen zurückzugeben, definieren wir die kompakte Funktion:

```
static int min( int a, int b ) { return a < b ? a : b; }
static int max( int a, int b ) { return a > b ? a : b; }
```

Beispiel Der Absolutwert einer Zahl wird zurückgegeben durch

```
x >= 0 ? x : -x
```

Beispiel Der Groß-/Kleinbuchstabe im ASCII-Alphabet soll zurückgegeben werden. Dabei ist `c` vom Typ `char`.

```
// Konvertierung in Großbuchstaben

c = (char)(Character.isLowerCase(c) ? (c-'a'+'A' ) : c);

// Konvertierung in Kleinbuchstaben

c = (char)(Character.isUpperCase(c) ? (c-'A'+'a' ) : c);
```

Da diese Umwandlung nur für die 26 ASCII-Buchstaben funktioniert, ist es besser, Bibliotheksmethoden für alle Unicode-Zeichen zu verwenden. Dazu dienen die Funktionen `toUpperCase()` und `toLowerCase()`.

Beispiel Es soll eine Zahl `n`, die zwischen 0 und 15 liegt, zur Hexadezimalzahl konvertiert werden.

```
(char)( (n < 10) ? ('0' + n) : ('a' - 10 + n ) )
```

Einige Fallen

Die Anwendung des trinären Operators führt schnell zu schlecht lesbaren Programmen und sollte daher vorsichtig eingesetzt werden. In C(++) führt die unbeabsichtigte Mehrfachauswertung in Makros zu schwer auffindbaren Fehlern. Gut, dass uns das in Java nicht passieren kann. Durch ausreichende Klammerung muss sichergestellt werden, dass die Ausdrücke auch in der beabsichtigten Reihenfolge ausgewertet werden. Im Gegensatz zu den meisten Operatoren ist der Bedingungsoperator rechtsassoziativ. (Die Zuweisung ist ebenfalls rechtsassoziativ.) Die Anweisung

```
b1 ? a1 : b2 ? a2 : a3
```

ist demnach gleichbedeutend mit

```
b1 ? a1 : ( b2 ? a2 : a3 )
```

Beispiel Wollen wir eine Funktion schreiben, die für eine Zahl `n` abhängig vom Vorzeichen -1, 0 oder 1 liefert, lösen wir das Problem mit geschachteltem trinären Operator.

```
int sign( int n )
{
  return (n < 0) ? -1 : (n > 0) ? 1 : 0;
}
```

Der Bedingungsoperator ist kein lvalue

Der trinäre Operator liefert als Ergebnis einen Ausdruck zurück, der auf der rechten Seite einer Zuweisung verwendet werden kann. Da er rechts vorkommt, nennt er sich auch *rvalue*. Er lässt sich nicht derartauf der linken Seite einer Zuweisung einsetzen, dass er eine Variable auswählt, der ein Wert zugewiesen wird.

Beispiel Folgende Anwendung des trinären Operators ist in Java **nicht** möglich:[34]

```
((richtung>=0) ? hoch : runter) = true;
```

2.9.7 Überladenes Plus für Strings

Obwohl sich in Java die Operatoren fast alle auf primitive Datentypen beziehen, gibt es doch eine bemerkenswerte Verwendung des Plus-Operators. Objekte vom Typ `String` können durch den Plus-Operator mit anderen Strings verbunden werden. Dies wurde in Java eingeführt, da ein Aneinanderhängen von Zeichenketten oft benötigt wird. Im Kapitel über die verschiedenen Klassen wird `String` noch etwas präziser dargestellt. Insbesondere werden die Gründe dargelegt, die zur Einführung eines String-Objekts auf der einen Seite, aber auch zur Sonderbehandlung in der Sprachdefinition auf der anderen Seite führten.

Listing 2.11 HelloName.java

```
// Ein Kleines ‚Trallala'-Programm in Java

class HelloName
{
  public static void main( String args[] )
  {
    // Zwei Strings deklarieren
    String name,
           intro;

    // Ersetze "Ulli" durch deinen Namen
```

34 In C(++) kann dies durch `*((Bedingung) ? &a : &b) = Ausdruck;` über Pointer gelöst werden.

```
    name = "Ulli";

    // Nun die Ausgabe
    intro = "Tri Tra Trallala \"Trullala\", sage " + name;
    System.out.println( intro );
  }
}
```

Nachdem ein String `intro` aus verschiedenen Objekten zusammengesetzt wurde, läuft die Ausgabe auf dem Bildschirm über die Funktion `println()` ab.

```
Tri Tra Trallala "Trullala", sage Ulli
```

Die Funktion schreibt den String auf die Konsolenausgabe und setzt hinter die Zeile noch einen Zeilenvorschub. (Obwohl das `\n` nicht unbedingt plattformunabhängig ist, wollen wir es trotzdem nutzen.[35]) Das Objekt `System.out` definiert den Ausgabekanal.

Beispiel Da der Plus-Operator für Zeichenketten streng von links nach rechts geht, bringt das mit eingebetteten arithmetischen Ausdrücken mitunter Probleme. Diese müssen dann geklammert werden, wie im Folgenden zu sehen:

```
"Ist 1 größer als 2? " + (1>2 ? "nein" : "ja");
```

Wäre der Ausdruck um den Bedingungsoperator nicht geklammert, dann würde der Plus-Operator den Ausdruck `1>2` auswerten, in einen String umwandeln und diesen an die erste Zeichenkette anhängen. Jetzt käme das Fragezeichen. Dies ist aber nur für boolesche Werte erlaubt, aber links stände die Zeichenkette. Das wäre ein Fehler.

2.10 Einfache Benutzereingaben

Bei den ersten eigenen Programmen möchte jeder neben der Ausgabe auch eine Eingabe in Java realisieren, damit ein Programm auch Benutzereingaben verarbeiten kann. Der Weg über die Befehlszeile ist dabei steinig, da Java eine Eingabe nicht so einfach wie eine Ausgabe vorsieht. Wer dennoch auf Benutzereingaben reagieren möchte, der kann dies über einen grafischen Eingabedialog `JOptionPane` realisieren.

Listing 2.12 InputWithDialog.java

```
import javax.swing.*;

class InputWithDialog
{
  public static void main( String args[] )
  {
    String s = JOptionPane.showInputDialog( "Wo kommst du denn wech?" );

    System.out.println( "Aha, du kommst aus " + s );
```

35 Besser ist es, das Absatzendezeichen aus der Systemeigenschaft zu nehmen. Dazu dient die Anweisung `System.getProperty("line.separator")`, die einen String liefert.

```
    System.exit( 0 );                          // Beendet das Programm
  }
}
```

Soll die Zeichenkette in eine Zahl konvertiert werden, dann können wir die Methode `parseInt()` nutzen.

Beispiel Zeige einen Eingabedialog an, der zur Zahleneingabe auffordert. Quadriere die eingelesene Zahl und gib sie auf dem Bildschirm aus.

```
String s = JOptionPane.showInputDialog( "Bitte Zahl eingeben" );
int i = Integer.parseInt( s );
System.out.println( i * i );
```

Fehler müssten zusätzlich in einen `try`/`catch`-Block gesetzt werden, da `parseInt()` eine `NumberFormatException` auslöst, wenn Buchstaben zur Umwandlung anstehen.

Beispiel Es soll ein einzelnes Zeichen eingelesen werden.

```
String s = JOptionPane.showInputDialog( "Bitte Zeichen eingeben" );
char c = 0;
if ( s != null && s.length() > 0 )
  c = s.charAt( 0 );
```

Beispiel Ein Wahrheitswert soll eingelesen werden. Dieser Wahrheitswert soll vom Benutzer als Zeichenkette `true` oder `false` beziehungsweise 1 oder 0 eingegeben werden.

```
String s = JOptionPane.showInputDialog( "Bitte Wahrheitswert eingeben" );
boolean buh;
if ( s != null )
  if (s.equals("0") || s.equals("false") )
    buh = false;
  else if (s.equals("1") || s.equals("true") )
    buh = true;
```