

Vorkurs/Softwareentwicklung

Eine kompakte Vorbereitung der Softwareentwicklung des Informatikstudiums

Von Fabio Hellmann

13. September 2015

Fachschaft 07

Auf zugrunde Lage von „Java ist auch eine Insel“ von Christian Ullenboom



1308 Seiten, 10., aktualisierte Auflage 2011, gebunden, mit DVD

ISBN 978-3-8362-1802-3

<http://openbook.rheinwerk-verlag.de/javainsel/>

Inhaltsverzeichnis

1	Einführung	4
1.1	Worum geht's im Vorkurs Softwareentwicklung?	4
1.2	An wen richtet sich dieser Kurs?	4
2	Java ist auch eine Sprache	4
2.1	Historischer Hintergrund	4
2.2	Warum Java gut ist: die zentralen Eigenschaften	6
2.2.1	Plattformunabhängigkeit.....	6
2.2.2	Java ist verbreitet und bekannt.....	7
2.2.3	Wofür sich Java weniger eignet.....	7
2.3	Die Installation der Java Platform Standard Edition (Java SE).....	8
2.3.1	Die Java SE von Oracle	8
2.3.2	Download des JDK	9
2.3.3	Java SE unter Windows installieren.....	10
2.4	Das erste Programm compilieren und testen	12
2.4.1	Ein Quadratzahlen-Programm	12
2.4.2	Der Compilerlauf	13
2.4.3	Die Laufzeitumgebung.....	15
3	Imperative Sprachkonzepte	15
3.1	Elemente der Programmiersprache Java	15
3.1.1	Bezeichner	15
3.1.2	Literale	16
3.1.3	Reservierte Schlüsselwörter	17
3.1.4	Kommentare.....	17
3.2	Von der Klasse zur Anweisung.....	18
3.2.1	Was sind Anweisungen?	18
3.2.2	Klassendeklaration	18
3.2.3	Die Reise beginnt am main()	19
3.2.4	Der erste Methodenaufruf: println().....	19
3.2.5	Die API-Dokumentation.....	19
3.2.6	Ausdrücke	20
3.2.7	Erste Idee der Objektorientierung.....	21
3.2.8	Modifizierer	21
3.3	Datentypen, Typisierung, Variablen und Zuweisungen	22

3.3.1	Primitive Datentypen im Überblick	23
3.3.2	Variablendeklarationen	24
3.3.3	Initialisierung von lokalen Variablen	26
3.4	Ausdrücke, Operanden und Operatoren.....	26
3.4.1	Zuweisungsoperator.....	27
3.4.2	Arithmetische Operatoren	28
3.4.3	Unäres Minus und Plus.....	30
3.4.4	Die relationalen Operatoren und die Gleichheitsoperatoren	30
3.4.5	Logische Operatoren: Nicht, Und, Oder, Xor.....	31
3.5	Bedingte Anweisungen oder Fallunterscheidungen	31
3.5.1	Die if-Anweisung.....	32
3.5.2	Die Alternative mit einer if-else-Anweisung wählen.....	33
3.6	Schleifen	33
3.6.1	Die while-Schleife	34
3.6.2	Die for-Schleife	35
3.7	Methoden einer Klasse.....	37
3.7.1	Bestandteil einer Methode	37
3.7.2	Aufruf einer Methode	38
3.7.3	Methoden ohne Parameter deklarieren	38
3.7.4	Statische Methoden (Klassenmethoden)	39
3.7.5	Parameter, Argument und Wertübergabe	40
3.7.6	Methoden mit Rückgaben	41
3.7.7	Rekursive Methoden	41
3.7.8	Die Türme von Hanoi.....	42

1 Einführung

1.1 Worum geht's im Vorkurs Softwareentwicklung?

Jeder Studierende tut sich in unterschiedlichen Fächern schwer bzw. leicht. Es gibt jedoch Fächer, in denen ist die Durchfallquote bereits über Jahre überdurchschnittlich hoch. Bei einem dieser Fächer mit besonders hoher Durchfallquote handelt es sich um das Fach Softwareentwicklung I. Aus diesem Grund haben wir als Fachschaft es uns, so wie bereits einige vor uns, zur Aufgabe gemacht, dem Ganzen ein wenig entgegen zu steuern und zu versuchen, euch die Grundlagen welche für dieses Fach von Nöten sind, zu vermitteln.

1.2 An wen richtet sich dieser Kurs?

Dieser Kurs richtet sich an all jene,

- welche noch keinerlei Vorkenntnisse in der Programmierung besitzen.
- deren Informatikkurs aus der Schule schon etwas länger her ist.
- und an alle die der Meinung sind, eine Wiederholung schadet nie.

2 Java ist auch eine Sprache¹

»Wir produzieren heute Informationen en masse, so wie früher Autos.«
– John Naisbitt (*1929)

Nach fast 20 Jahren hat sich Java als Plattform etabliert. Über 9 Millionen Softwareentwickler verdienen weltweit mit der Sprache ihre Brötchen, 3 Milliarden Mobiltelefone führen Java-Programme aus,[5](So verkündet es Thomas Kurian auf der JavaOne 2010-Konferenz.

Auch <http://www.oracle.com/us/corporate/press/193190.>) 1,1 Milliarden Desktops und alle Blu-ray-Player. Es gibt 10 Millionen Downloads von Oracles Laufzeitumgebung in jeder Woche, was fast 1 Milliarde Downloads pro Jahr ergibt.

Dabei war der Erfolg nicht unbedingt vorhersehbar. Java[6](*Just Another Vague Acronym* (etwa »bloß ein weiteres unbestimmtes Akronym«)) hätte einfach nur eine schöne Insel, eine reizvolle Wandfarbe oder eine Pinte mit brasilianischen Rhythmen in Paris sein können, so wie Heuschrecken einfach nur grüne Hüpfen hätten bleiben können. Doch als robuste objektorientierte Programmiersprache mit einem großen Satz von Bibliotheken ist Java als Sprache für Softwareentwicklung im Großen angekommen und im Bereich plattformunabhängiger Programmiersprachen konkurrenzlos.

2.1 Historischer Hintergrund

In den 1970er-Jahren wollte Bill Joy eine Programmiersprache schaffen, die alle Vorteile von *MESA* und *C* vereinigen sollte. Diesen Wunsch konnte sich Joy zunächst nicht erfüllen, und erst Anfang der 1990er-Jahre beschrieb er in dem Artikel »Further«, wie eine neue objektorientierte Sprache aussehen könnte; sie sollte in den Grundzügen auf C++ aufbauen. Erst später wurde ihm bewusst, dass C++ als Basissprache ungeeignet und für große Programme unhandlich ist.

Zu jener Zeit arbeitete James Gosling am *SGML*-Editor *Imagination*. Er entwickelte in C++ und war mit dieser Sprache ebenfalls nicht zufrieden. Aus diesem Unmut heraus entstand die neue Sprache *Oak*. Der Name fiel Gosling ein, als er aus dem Fenster seines Arbeitsraums schaute – und eine Eiche

¹ Ab hier werden Ausschnitte aus dem Buch „Java ist auch eine Insel“ von Christian Ullenboom (<http://openbook.rheinwerk-verlag.de/javainsel/>) verwendet.

erblickte (engl. *oak*), doch vielleicht ist das nur eine Legende, denn Oak steht auch für *Object Application Kernel*. Patrick Naughton startete im Dezember 1990 das Green-Projekt, in das Gosling und Mike Sheridan involviert waren. Überbleibsel aus dem Green-Projekt ist der *Duke*, der zum bekannten Symbol wurde.[7](Er sieht ein bisschen wie ein Zahn aus und könnte deshalb auch die Werbung eines Zahnarztes sein. Das Design stammt übrigens von Joe Palrang.)



Abbildung 1.1: Der Duke, Symbol für Java

Die Idee hinter diesem Projekt war, Software für interaktives Fernsehen und andere Geräte der Konsumelektronik zu entwickeln. Bestandteile dieses Projekts waren das Betriebssystem Green-OS, Goslings Interpreter Oak und einige Hardwarekomponenten. Joy zeigte den Mitgliedern des Green-Projekts seinen Further-Aufsatz und begann mit der Implementierung einer grafischen Benutzeroberfläche. Gosling schrieb den Original-Compiler 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 es im Herbst 1992 intern vorstellte. Der ehemalige Sun-Chef Scott McNealy (der nach der Übernahme von Oracle im Januar 2010 das Unternehmen verließ) 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, dass Time Warner ein System für Set-Top-Boxen suchte (Set-Top-Boxen sind elektronische Geräte für Endbenutzer, die etwa an einen Fernseher angeschlossen werden). 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 *W3*). Die Programmiersprache sollte Programmcode über das Netzwerk empfangen können, und fehlerhafte Programme sollten keinen Schaden anrichten. 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 erhielt den Namen *Java*, da der Name Oak, wie sich später herausstellte, aus Gründen des Copyrights nicht verwendet werden konnte: Eine andere Programmiersprache schmückte sich bereits mit diesem Namen. Nach der Überlieferung fiel die Entscheidung für den Namen Java in einem Coffeeshop. In Java führte Patrick Naughton den Prototyp des Browsers *WebRunner* vor, der an einem Wochenende entstanden sein soll. Nach geringfügiger Überarbeitung durch Jonathan Payne wurde der Browser *HotJava* getauft und im Mai auf der SunWorld '95 der Öffentlichkeit vorgestellt.

Zunächst konnten sich nur wenige Anwender mit HotJava anfreunden. So war es ein 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 Siegeszug.

2.2 Warum Java gut ist: die zentralen Eigenschaften

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 objektorientiertes Programmieren ermöglicht, können Entwickler moderne und wiederverwertbare Softwarekomponenten programmieren.

Zum Teil wirkt Java sehr konservativ, aber das liegt daran, dass die Sprachdesigner nicht alles das sofort einbauen, was im Moment gerade hipp ist (XML-Literale sind so ein Beispiel). Java nahm schon immer das, was sich in anderen Programmiersprachen als sinnvoll und gut herausgestellt hat, in den Sprachkern auf, vermied es aber, Dinge aufzunehmen, die nur von sehr wenigen Entwicklern eingesetzt werden bzw. die öfter zu Fehlern führen. In den Anfängen stand C++ als Vorbild da, heute schielt Java auf C# und Skriptsprachen.

Einige der zentralen Eigenschaften wollen wir uns im Folgenden anschauen und dabei auch zentrale Begriffe und Funktionsweisen beleuchten.

2.2.1 Plattformunabhängigkeit

Eine zentrale Eigenschaft von Java ist seine Plattformunabhängigkeit bzw.

Betriebssystemunabhängigkeit. Diese wird durch zwei zentrale Konzepte erreicht. Zum einen bindet sich Java nicht an einen bestimmten Prozessor oder eine bestimmte Architektur, sondern der Compiler generiert Bytecode, den eine Laufzeitumgebung dann abarbeitet. Zum anderen abstrahiert Java von den Eigenschaften eines konkreten Betriebssystems, schafft etwa eine Schnittstelle zum Ein-/Ausgabesystem oder eine API für grafische Oberflächen. Entwickler programmieren immer gegen eine Java-API aber nie gegen die API der konkreten Plattform, etwa die Windows- oder Unix-API. Die Java-Laufzeitumgebung bildet Aufrufe etwa auf Dateien für das jeweilige System ab, ist also Vermittler zwischen den Java-Programmen und der eigentlichen Betriebssystem-API.

Zwar ist das Konzept einer plattformneutralen Programmiersprache schon recht alt, doch erst in den letzten 10 Jahren kam mehr und mehr hinzu. Neben Java sind plattformunabhängige Programmiersprachen und Laufzeitumgebungen .NET-Sprachen wie C# auf der CLR (*Common Language Runtime* – entspricht der Java VM), Perl, Python oder Ruby. Plattformunabhängigkeit ist schwer, denn die Programmiersprache und ein Bytecode produzierender Compiler ist nur ein Teil – der größere Teil ist die Laufzeitumgebung und eine umfangreiche API. Zwar ist auch C an sich eine portable Sprache, und ANSI C-Programme lassen sich von jedem C-Compiler auf jedem Betriebssystem mit Compiler übersetzen, aber das Problem sind die Bibliotheken, die über ein paar simple Dateioperationen nicht hinauskommen.

In Java 7 ändert sich die Richtung etwas, was sich besonders an der neuen API für die Dateisystemunterstützung ablesen lässt. Vor Java 7 war die Datei-Klasse so aufgebaut, dass die Semantik gewisser Operationen nicht ganz genau spezifiziert war und auf diese Weise sehr plattformabhängig war. Es gibt aber in der Datei-Klasse keine Operation, die nur auf einer Plattform zur Verfügung steht und andere Plattformen ausschließt. Das Credo lautete immer: Was nicht auf allen Plattformen existiert, kommt nicht in die Bibliothek.[11](Es gibt sie durchaus, die Methoden, die nur zum Beispiel auf Windows zur Verfügung stehen. Aber dann liegen sie nicht in einem java- oder javax-Paket, sondern in einem internen Paket.) Mit Java 7 gibt es einen Wechsel: Nun sind plattformspezifische

Dateieigenschaften zugänglich. Es bleibt abzuwarten, ob in der Zukunft in anderen API-Bereichen – vielleicht bei grafischen Oberflächen – noch weitere Beispiele hinzukommen.

2.2.2 Java ist verbreitet und bekannt

Unabhängig von der Leistungsfähigkeit einer Sprache zählen am Ende doch nur betriebswirtschaftliche Faktoren: Wie schnell und billig lässt sich ein vom Kunden gewünschtes System bauen, und wie stabil und änderungsfreundlich ist es? Dazu kommen Fragen wie: Wie sieht der Literaturmarkt aus, wie die Ausbildungswege, woher bekommt ein Team einen Entwickler oder Consultant, wenn es brennt? Dies sind nicht unbedingt Punkte, die Informatiker beim Sprachvergleich auf die erste Stelle setzen, sie sind aber letztendlich für den Erfolg einer Software-Plattform entscheidend. Fast jede Universität lehrt Java, und mit Java ist ein Job sicher. Konferenzen stellen neue Trends vor und schaffen Trends. Diese Kette ist nicht zu durchbrechen, und selbst wenn heute eine neue Super-Sprache mit dem Namen »Bali« auftauchen würde, würde es Jahre dauern, bis ein vergleichbares System geschaffen wäre. Wohlgemerkt: Das sagt nichts über die Innovations- oder Leistungsfähigkeit aus, nur über die Marktsättigung, aber dadurch wird Java eben für so viele interessant.

2.2.3 Wofür sich Java weniger eignet

Java ist als Programmiersprache für allgemeine Probleme entworfen worden und deckt große Anwendungsgebiete ab (*general-purpose language*). Das heißt aber auch, dass es für ausreichend viele Anwendungsfälle deutlich bessere Programmiersprachen gibt, etwa im Bereich Skripting, wo die Eigenschaft, dass jedes Java-Programm mindestens eine Klasse und eine Methode benötigt, eher störend ist, oder im Bereich von automatisierter Textverarbeitung, wo andere Programmiersprachen eleganter mit regulären Ausdrücken arbeiten können.

Auch dann, wenn extrem maschinen- und plattformabhängige Anforderungen bestehen, wird es in Java umständlich. Java ist plattformunabhängig entworfen worden, sodass alle Methoden auf allen Systemen lauffähig sein sollen. Sehr systemnahe Eigenschaften wie die Taktfrequenz sind nicht sichtbar, und sicherheitsproblematische Manipulationen wie der Zugriff auf bestimmte Speicherzellen (das PEEK und POKE) sind ebenso untersagt. Hier ist eine bei Weitem unvollständige Aufzählung von Dingen, die Java standardmäßig nicht kann:

- CD auswerfen
- Bildschirm auf der Textkonsole löschen, Cursor positionieren und Farben setzen
- auf niedrige Netzwerk-Protokolle wie ICMP zugreifen
- Microsoft Office fernsteuern
- Zugriff auf USB[25](Eigentlich sollte es Unterstützung für den Universal Serial Bus geben, doch Sun hat hier – wie leider auch an anderer Stelle – das Projekt JSR-80: Java USB API nicht weiterverfolgt.) oder Firewire

Aus den genannten Nachteilen, dass Java nicht auf die Hardware zugreifen kann, folgt, dass die Sprache nicht so ohne Weiteres für die Systemprogrammierung eingesetzt werden kann. Treibersoftware, die Grafik-, Sound- oder Netzwerkkarten anspricht, lässt sich in Java nur über Umwege realisieren. Genau das Gleiche gilt für den Zugriff auf die allgemeinen Funktionen des

Betriebssystemen, die Windows, Linux oder ein anderes System bereitstellt. Typische System-Programmiersprachen sind C(++) oder Objective-C.

Aus diesen Beschränkungen ergibt sich, dass Java eine hardwarenahe Sprache wie C(++) nicht ersetzen kann. Doch das muss die Sprache auch nicht! Jede Sprache hat ihr bevorzugtes Terrain, und Java ist eine allgemeine Applikationsprogrammiersprache; C(++) darf immer noch für Hardwaretreiber und virtuelle Java-Maschinen erhalten.

Soll ein Java-Programm trotzdem systemnahe Eigenschaften nutzen – und das kann es mit entsprechenden Bibliotheken ohne Probleme –, bietet sich zum Beispiel der *native Aufruf* einer Systemfunktion an. Native Methoden sind Unterprogramme, die nicht in Java implementiert werden, sondern in einer anderen Programmiersprache, häufig in C(++). In manchen Fällen lässt sich auch ein externes Programm aufrufen und so etwa die Windows-Registry manipulieren oder Dateirechte setzen. Es läuft aber immer darauf hinaus, dass die Lösung für jede Plattform immer neu implementiert werden muss.

2.3 Die Installation der Java Platform Standard Edition (Java SE)

Die folgende Anleitung beschreibt, woher wir Oracles Java SE-Implementierung beziehen können und wie die Installation verläuft.

2.3.1 Die Java SE von Oracle

Es gibt unterschiedliche Möglichkeiten, in den Besitz der Java SE zu kommen. Wer einen schnellen Zugang zum Internet hat, kann es sich von den Oracle-Seiten herunterladen. Nicht-Internet-Nutzer oder Anwender ohne schnelle Verbindungen finden Entwicklungsversionen häufig auch auf DVDs, wie etwa der DVD in diesem Buch.

JDK und JRE

In der Java SE-Familie gibt es verschiedene Ausprägungen: das JDK und das JRE. Da diejenigen, die Java-Programme nur laufen lassen möchten, nicht unbedingt alle Entwicklungstools benötigen, hat Oracle zwei Pakete geschnürt:

- Das *Java SE Runtime Environment* (JRE) enthält genau das, was zur Ausführung von Java-Programmen nötig ist. Die Distribution umfasst nur die JVM und Java-Bibliotheken, aber weder den Quellcode der Java-Bibliotheken noch Tools.
- Mit dem *Java Development Kit* (JDK) lassen sich Java SE-Applikationen entwickeln. Dem JDK sind Hilfsprogramme beigelegt, die für die Java-Entwicklung nötig sind. Dazu zählen der essenzielle Compiler, aber auch andere Hilfsprogramme, etwa zur Signierung von Java-Archiven oder zum Start einer Management-Konsole. In den Versionen Java 1.2, 1.3 und 1.4 heißt das JDK *Java 2 Software Development Kit* (J2SDK), kurz *SDK*, ab Java 5 heißt es wieder JDK.

Das JRE und JDK von Oracle sind beide gratis erhältlich. Das Projekt ist selbst extrem komplex und umfasst (Stand 2009) etwa 900.000 Zeilen C++-Code.[39](http://lvm.org/devmtg/2009-10/Cifuentes_ParfaitBugChecker.pdf)

2.3.2 Download des JDK

Oracle bietet auf der Webseite <http://www.oracle.com/technetwork/java/javase/downloads/> das Java SE und andere Dinge zum Download an. Oracle bündelt die Implementierung in unterschiedliche Pakete:

- JDK
- JRE
- JDK mit Java EE
- JDK mit der NetBeans IDE



Abbildung 1.4: Die Java SE-Download-Seite von Oracle

Wir entscheiden uns für das pure JDK, denn es enthält einige Entwicklungstools, die wir später benötigen (auch wenn für Eclipse und einfache Programme prinzipiell das JRE ausreicht)[40](Eclipse bringt einen eigenen Java-Compiler mit, daher ist der Compiler des JDK nicht nötig. Doch das JDK bringt auch die Quellen der Java-Bibliotheken mit, was sehr nützlich ist, denn aus den Quellen wird direkt die API-Dokumentation generiert. Sie muss daher nicht extra bezogen werden.).

Ein Klick auf JDK Download führt zur

Seite <http://www.oracle.com/technetwork/java/javase/downloads/java-se-jdk-7-download-432154.html>, die direkt zu Installationspaketen für folgende Systeme führt:

- Microsoft Windows für jeweils 32- und 64-Bit-Systeme
- Solaris SPARC 32 und 64 Bit, Solaris x86, Solaris x64
- Linux x86 und Linux x64

Für Windows x86 ist es der Link <http://download.oracle.com/otn-pub/java/jdk/7/jdk-7-windows-i586.exe>. Ein Download ist jedoch erst dann möglich, wenn die Lizenzbestimmungen (Oracle Binary Code License Agreement for Java SE) bestätigt wurden.

Download der Dokumentation

Die API-Dokumentation der Standardbibliothek und Tools ist kein Teil des JDK (bei einer Größe des JDK von fast 76 MiB ist eine Trennung sinnvoll, denn die Dokumentation selbst umfasst etwa die gleiche Größe). Die Hilfe kann online unter <http://www.tutego.de/go/javaapi/> eingesehen oder als Zip-Datei extra bezogen und lokal ausgepackt werden. Das komprimierte Archiv ist auf der DVD oder unter <http://www.oracle.com/technetwork/java/index-jsp-142903.html#documentation> erhältlich. Ausgepackt ist die API-Dokumentation eine Sammlung von HTML-Dateien. Unter <http://www.allimant.org/javadoc/index.php> findet sich die Hilfe auch im HTMLHelp- und WinHelp-Format. Diese Microsoft-Formate erleichtern die Suche in der Dokumentation.

2.3.3 Java SE unter Windows installieren

Die ausführbare Datei jdk-7-windows-i586.exe ist das Installationsprogramm. Es installiert die ausführbaren Programme wie Compiler und Interpreter sowie die Bibliotheken, Quellcodes und auch Beispielpprogramme. Voraussetzung für die Installation sind genügend Rechte, ein paar MiB Plattenspeicher und die Windows-Service-Packs. Quellcodes und Demos müssen nicht unbedingt installiert sein. Die Datei jdk-7-windows-i586.exe ist die einzige, die für Java zwingend installiert werden muss; alles andere sind Extras, wie grafische Entwicklungsumgebungen.

Schritt-für-Schritt-Installation

Gehen wir nun Schritt für Schritt durch die Installation.



Abbildung 1.5: Der Startbildschirm

Nach dem Klick auf Next fragt der Installer nach den zu installierenden Komponenten. Zuerst wird das JDK, dann das JRE installiert.



Abbildung 1.6: Auswahl der Komponenten

Ein Klick auf Change, und wir können das Installationsverzeichnis des JDK ändern. Mit Next beginnt die Installation des JDK.

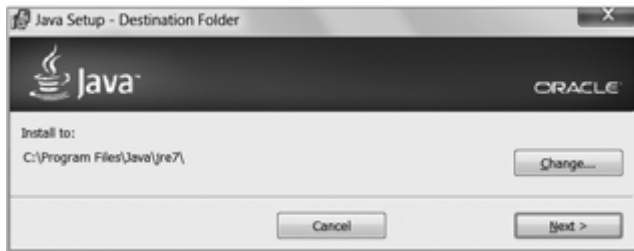


Abbildung 1.7: Installationsverzeichnis für das JRE bestimmen.

Nach der Installation des JDK folgt das JRE. Auch hier lässt sich das Verzeichnis ändern. Wir aktivieren Next. Anschließend folgt ein Dialog, mit dem wir uns für Updates registrieren lassen können. Finish beendet die Installation.



Abbildung 1.8: Die Installation wurde erfolgreich beendet.

Nach der abgeschlossenen Installation können wir unter Windows – ohne Änderung des Installationsverzeichnisses – im Dateibaum unter [C:\Program Files\Java](#) bzw. C:\Programme\Java (oder C:\Program Files (x86)\Java bzw. C:\Program Files (x64)\Java je nach Windows-Version) die beiden Ordner jdk1.7.0 und jre ausmachen (für die folgenden Beispiele geben wir immer den Pfad C:\Program Files\Java\jdk1.7.0 an).

Registry-Einträge unter Windows

Unter Windows wird in der Registry ein Zweig HKEY_CURRENT_USER\Software\JavaSoft angelegt, in dem unter anderem Preferences-Einstellungen gespeichert werden. Weiterhin gibt es einen Zweig HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths\javaws.exe mit dem Wert C:\Program Files\Java\jre7\bin\javaws.exe und dem zusätzlichen Schlüssel Path, der mit C:\Program Files\Java\jre7\bin belegt wird.

Ungünstig ist, dass der Installer nicht überprüft, ob eine alte Version schon vorhanden ist – er überschreibt die Registry-Einträge einfach. Die Lösungen sind nicht wirklich befriedigend: Entweder wird die Registry später restauriert oder eine alte Java-Version wird noch einmal darüber installiert.

Java im C:\Windows\System32\-Pfad

Der Installer setzt unter Windows die ausführbare Datei `java.exe` in das System-Verzeichnis `C:\Windows\System32\`. Das Programm liest die Registry aus (wie gerade beschrieben) und startet die dort eingetragene JVM. Wer also den Standard ändern möchte, der muss die Registry-Schlüssel umbiegen.

Der Grund für die Installation an dieser zentralen Systemstelle ist, dass das Verzeichnis Teil des Windows-Standard-Suchpfades ist und dass wir nach der Installation von der Kommandozeile aus über `java` sofort Programme starten können. Der Installer könnte grundsätzlich auch die `PATH`-Variable erweitern, doch Oracle wählte diesen Weg nicht. Im System-Ordner ist auch nur `java` eingetragen, nicht aber `javac`, weshalb Entwickler doch oft die `PATH`-Variable auf das `bin`-Verzeichnis des JDK setzen.

Programme im bin-Verzeichnis

Im `bin`-Verzeichnis des JDK (`C:\Programme\Java\jdk1.7.0\bin`) sind zusätzliche Entwicklungswerkzeuge und Java-Quellen untergebracht, die das JRE nicht enthält. Leider wird aber auch vieles doppelt installiert;^[41] (Das JRE gibt es so zum Beispiel unter `C:\Program Files\Java\jdk1.7.0\jre` und auch unter `C:\Program Files\Java\jre7`. Daraus folgt aber auch, dass zusätzliche Bibliotheken auch an zwei Stellen installiert werden müssen.) Oracle betrachtet das JRE nicht als Teilmenge des JDK.

Der JDK-Ordner hat nicht viele Verzeichnisse und Dateien. Die wichtigsten sind:

- `bin`: Hier befinden sich unter anderem der Compiler *javac* und der Interpreter *java*.
- `jre`: die eingebettete Laufzeitumgebung
- `demo` (falls installiert): Diverse Unterverzeichnisse enthalten Beispiel-Programme. Ein interessantes Demo finden wir unter `jfc/Java2D` (ein Doppelklick auf die `.jar`-Datei startet es) und online unter <http://download.java.net/javadesktop/swingset3/SwingSet3.jnlp>.
- `src.zip` (falls installiert): Das Archiv enthält den Quellcode der öffentlichen Bibliotheken. Entwicklungsumgebungen wie Eclipse und NetBeans binden die Quellen automatisch mit ein, sodass sie leicht über einen Tastendruck zugänglich sind.

2.4 Das erste Programm compilieren und testen

Nachdem wir die grundlegenden 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.« In diesem Abschnitt nutzen wir den Java-Compiler und Interpreter von der Kommandozeile. Wer gleich eine ordentliche Entwicklungsumgebung wünscht, der kann problemlos diesen Teil überspringen und bei den IDEs fortfahren.

2.4.1 Ein Quadratzahlen-Programm

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 Methoden enthält. Alle Methoden in

einer objektorientierten Programmiersprache wie Java müssen in Klassen platziert werden. Die erste Methode, `quadrat()`, bekommt als Übergabeparameter eine ganze Zahl und berechnet daraus die Quadratzahl, die sie anschließend zurückgibt. Eine weitere Methode übernimmt die Ausgabe der Quadratzahlen bis zu einer vorgegebenen Grenze. Die Methode bedient sich dabei der Methode `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 Methode `ausgabe()` auf.

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 );
    }
}
```

2.4.2 Der Compilerlauf

Der Quellcode eines Java-Programms lässt sich so allein nicht ausführen. Ein spezielles Programm, der *Compiler* (auch *Übersetzer* genannt), transformiert das geschriebene Programm in eine andere Repräsentation. Im Fall von Java erzeugt der Compiler die DNA jedes Programms, den Bytecode.

Wir wechseln zur Eingabeaufforderung (Konsole) und in das Verzeichnis mit dem Quellcode. Damit sich Programme übersetzen und ausführen lassen, müssen wir die Programme *javac* und *java* aus dem bin-Verzeichnis der JDK-Installation aufrufen.

Liegt die Quellcodedatei vor, übersetzt der Compiler sie in Bytecode.

C:\projekte>**javac** Quadrat.java

Alle Java-Klassen übersetzt *javac *.java*. Wenn die Dienstprogramme *javac* und *java* nicht im Suchpfad stehen, müssen wir einen kompletten Pfadnamen angeben – wie C:\Program Files\Java\jdk1.7.0\bin\javac *.java.

Die zu übersetzende Datei muss – ohne Dateiendung – so heißen wie die in ihr definierte öffentliche Klasse. Die Beachtung der Groß- und Kleinschreibung ist wichtig. Eine andere Endung, wie etwa .txt oder .jav, ist nicht erlaubt und führt zu einer Fehlermeldung:

```
C:\projekte>javac Quadrat.txt
```

```
Quadrat.txt is an invalid option or argument.
```

```
Usage: javac <options> <source files>
```

Der Compiler legt – vorausgesetzt, das Programm war fehlerfrei – die Datei Quadrat.class an. Diese enthält den Bytecode.

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 wir in der quadrat()-Methode einen Fehler ein (das Semikolon fällt der Löschtaste zum Opfer). Der Compilerdurchlauf meldet:

```
Quadrat.java:10: ';' expected.
```

```
    return n * n
```

```
        ^
```

```
1 error
```

Den Suchpfad für Windows setzen

Da es unpraktisch ist, bei jedem Aufruf immer den kompletten Pfad zur JDK-Installation anzugeben, lässt sich der Suchpfad erweitern, in dem die Shell nach ausführbaren Programmen sucht. Um die Pfade dauerhaft zu setzen, müssen wir die Umgebungsvariable PATH modifizieren. Für eine Sitzung reicht es, den bin-Pfad des JDK hinzuzunehmen. Wir setzen dazu in der Kommandozeile von Windows den Pfad jdk1.7.0\bin an den Anfang der Suchliste, damit im Fall von Altinstallationen immer das neue JDK verwendet wird:

```
set PATH=C:\Program Files\Java\jdk1.7.0\bin;%PATH%
```

Die Anweisung modifiziert die Pfad-Variable und legt einen zusätzlichen Verweis auf das bin-Verzeichnis von Java an.

Damit die Pfadangabe auch nach einem Neustart des Rechners noch verfügbar ist, müssen wir abhängig vom System unterschiedliche Einstellungen vornehmen.

- Ab Windows Vista aktiviere im Startmenü Systemsteuerung, und klicke dann auf System und Sicherheit. Aktiviere System und dann Erweiterte Systemeigenschaften. Es öffnet sich ein neuer Dialog mit einer Schaltfläche Umgebungsvariablen... Im unteren Teil, Systemvariablen, scrolle nach Path. Mit Bearbeiten... verändere den Eintrag, und füge dem Pfad hinter einem Semikolon das JDK-bin-Verzeichnis hinzu. Bestätige die Änderung mit OK, OK, OK.
- Unter Windows XP aktivieren wir den Dialog Systemeigenschaften unter Start • Einstellungen • Systemsteuerung • System. Unter dem Reiter Erweitert wählen wir die Schaltfläche Umgebungsvariablen, wo wir anschließend bei Systemvariablen die Variable Path auswählen und mit Bearbeiten verändern – natürlich können statt der Systemvariablen auch die lokalen Benutzereinstellungen modifiziert werden; da gibt es Path noch einmal. Hinter einem

Semikolon tragen wir den Pfad zum bin-Verzeichnis ein. Dann können wir den Dialog mit OK, OK, OK verlassen. War eine Eingabeaufforderung offen, bekommt sie von der Änderung nichts mit; ein neues Eingabeaufforderungsfenster muss geöffnet werden.

Weitere Hilfen gibt die Datei <http://tutego.de/go/installwindows>.

2.4.3 Die Laufzeitumgebung

Der vom Compiler erzeugte Bytecode ist kein üblicher Maschinencode für einen speziellen Prozessor, da Java als plattformunabhängige Programmiersprache entworfen wurde, die sich also nicht an einen physikalischen Prozessor klammert – Prozessoren wie Intel-, AMD- oder PowerPC-CPU's können mit diesem Bytecode nichts anfangen. Hier hilft eine Laufzeitumgebung weiter. Diese liest die Bytecode-Datei Anweisung für Anweisung aus und führt sie auf dem konkreten Mikroprozessor aus.

Der Interpreter java bringt das Programm zur Ausführung:

```
C:\projekte>java Quadrat
Quadrat(1) = 1
Quadrat(2) = 4
Quadrat(3) = 9
Quadrat(4) = 16
```

Als Argument bekommt die Laufzeitumgebung java den Namen der Klasse, die eine `main()`-Methode enthält und somit als ausführbar gilt. Die Angabe ist nicht mit der Endung `.class` zu versehen, da hier kein Dateiname, sondern ein Klassenname gefordert ist.

3 Imperative Sprachkonzepte

»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 kennenlernen, und dann ist der Weg frei für größere Programme.

3.1 Elemente der Programmiersprache Java

Wir wollen im Folgenden über das Regelwerk, die Grammatik und die Syntax der Programmiersprache Java sprechen und uns unter anderem über die Unicode-Kodierung, Tokens sowie Bezeichner Gedanken machen. Bei der Benennung einer Methode zum Beispiel dürfen wir aus einer großen Anzahl Zeichen wählen; der Zeichenvorrat nennt sich *Lexikalik*.

Die Syntax eines Java-Programms definiert die Tokens 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.

3.1.1 Bezeichner

Für Variablen (und damit Konstanten), Methoden, Klassen und Schnittstellen werden *Bezeichner* vergeben – auch *Identifizierer* (von engl. *identifier*) genannt –, die die entsprechenden Bausteine anschließend im Programm identifizieren. Unter Variablen sind dann

Daten verfügbar. Methoden sind die Unterprogramme 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 Unicode-Zeichensatz, und jedes Zeichen ist für die Identifikation wichtig.[50](Die Java-Methoden `Character.isJavaIdentifierStart()/isJavaIdentifierPart()` stellen auch fest, ob Zeichen Java-Identifizierer sind.) 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.

Beispiel

Im folgenden Java-Programm sind die Bezeichner fett und unterstrichen gesetzt.

```
class Application
{
    public static void main( String[] args )
    {
        System.out.println( "Hallo Welt" );
    }
}
```

Dass `String` fett und unterstrichen ist, hat seinen Grund, denn `String` ist eine Klasse und kein eingebauter Datentyp wie `int`. Zwar wird die Klasse `String` in Java bevorzugt behandelt – das Plus kann Zeichenketten zusammenhängen –, aber es ist immer noch ein Klassentyp.

3.1.2 Literale

Ein Literal ist ein konstanter Ausdruck. Es gibt verschiedene Typen von Literalen:

- die Wahrheitswerte `true` und `false`
- integrale Literale für Zahlen, etwa `122`
- Zeichenliterale, etwa `'X'` oder `'\n'`
- Fließkommaliterale, etwa `12.567` oder `9.999E-2`
- Stringliterale für Zeichenketten, wie `"Paolo Pinkas"`
- `null` steht für einen besonderen Referenztyp.

Beispiel

Im folgenden Java-Programm sind die beiden Literale fett und unterstrichen gesetzt.

```
class Application
{
    public static void main( String[] args )
    {
        System.out.println( "Hallo Welt" );
        System.out.println( 1 + 2 );
    }
}
```


3.1.3 Reservierte Schlüsselwörter

Bestimmte Wörter sind als Bezeichner nicht zulässig, da sie als *Schlüsselwörter* vom Compiler besonders behandelt werden. Schlüsselwörter bestimmen die »Sprache« eines Compilers.

Beispiel

Reservierte Schlüsselwörter sind im Folgenden fett und unterstrichen gesetzt.

```
class Application
{
    public static void main( String[] args )
    {
        System.out.println( "Hallo Welt" );
    }
}
```

3.1.4 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, Methoden 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 nach Monaten wir selbst) unsere Lösungsidee schnell nachvollziehen und später das Programm erweitern oder abändern können, werden *Kommentare* in den Quelltext geschrieben. Sie dienen nur den Lesern der Programme, haben aber auf die Abarbeitung keine Auswirkungen.

Unterschiedliche Kommentartypen

In Java gibt es zum Formulieren von Kommentaren drei Möglichkeiten:

- *Zeilenkommentare*: Sie beginnen mit zwei Schrägstrichen[53](In C++ haben die Entwickler übrigens das Zeilenkommentarzeichen // aus der Vor-Vorgängersprache BCPL wieder eingeführt, das in C entfernt wurde.) // und kommentieren den Rest einer Zeile aus. Der Kommentar gilt von diesen Zeichen an bis zum Ende der Zeile, also bis zum Zeilenumbruchzeichen.
- *Blockkommentare*: Sie kommentieren in /* */ Abschnitte aus. Der Text im Blockkommentar darf selbst kein */ enthalten, denn Blockkommentare dürfen nicht verschachtelt sein.
- *JavaDoc-Kommentare*: Das sind besondere Blockkommentare, die JavaDoc-Kommentare mit /** */ enthalten. Ein JavaDoc-Kommentar beschreibt etwa die Methode oder die Parameter, aus denen sich später die API-Dokumentation generieren lässt.

Schauen wir uns ein Beispiel an, in dem alle drei Kommentartypen vorkommen:

```

/*
 * Der Quellcode ist public domain.
 */
// Magic. Do not touch.
/**
 * @author Christian Ullenboom
 */
class DoYouHaveAnyCommentsToMake      // TODO: Umbenennen
{
    // When I wrote this, only God and I understood what I was doing
    // Now, God only knows
    public static void main( String[] args /* Kommandozeilenargument */ )
    {
    }
}

```

3.2 Von der Klasse zur Anweisung

Programme sind Ablauffolgen, die im Kern aus Anweisungen bestehen. Sie werden zu größeren Bausteinen zusammengesetzt, den Methoden, die wiederum Klassen bilden. Klassen selbst werden in Paketen gesammelt, und eine Sammlung von Paketen wird als Java-Archiv ausgeliefert.

3.2.1 Was sind Anweisungen?

Java zählt zu den imperativen Programmiersprachen, in denen der Programmierer die Abarbeitungsschritte seiner Algorithmen durch *Anweisungen* (engl. *statements*) vorgibt.

Anweisungen können unter anderem sein:

- Ausdrucksanweisungen, etwa für Zuweisungen oder Methodenaufrufe
- Fallunterscheidungen, zum Beispiel mit `if`
- Schleifen für Wiederholungen, etwa mit `for` oder `do-while`

3.2.2 Klassendeklaration

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 heißt *Kompilationseinheit* (engl. *compilation unit*) und deklariert eine Klasse mit ihren Methoden und Variablen.

Die nächsten Programmcodezeilen werden am Anfang etwas befremdlich wirken (wir erklären die Elemente später genauer). Die folgende Datei erhält den (frei wählbaren) Namen *Application.java*:

Listing 2.1: Application.java

```

public class Application
{
    public static void main( String[] args )
    {
        // Hier ist der Anfang unserer Programme
        // Jetzt ist hier Platz für unsere eigenen Anweisungen
        // Hier enden unsere Programme
    }
}

```

3.2.3 Die Reise beginnt am main()

Wir programmieren hier eine besondere Methode, die sich `main()` nennt. Die Schlüsselwörter davor und die Angabe in dem Paar runder Klammern hinter dem Namen müssen wir einhalten. Die Methode `main()` ist für die Laufzeitumgebung etwas ganz Besonderes, denn beim Aufruf des Java-Interpreters mit einem Klassennamen wird unsere Methode als Erstes ausgeführt.[54](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.) 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 produziert, obwohl die Methode selbst korrekt gebildet ist. Innerhalb von `main()` befindet sich ein Parameter mit dem Namen `args`. Der Name ist willkürlich gewählt, wir werden allerdings immer `args` verwenden.

3.2.4 Der erste Methodenaufruf: `println()`

In Java gibt es eine große Klassenbibliothek, die es Entwicklern erlaubt, Dateien anzulegen, Fenster zu öffnen, auf Datenbanken zuzugreifen, Web-Services aufzurufen und vieles mehr. Am untersten Ende der Klassenbibliothek stehen Methoden, die eine gewünschte Operation ausführen.

Eine einfache Methode ist `println()`. Sie gibt Meldungen auf dem Bildschirm (der Konsole) aus. Innerhalb der Klammern von `println()` können wir Argumente angeben. Die `println()`-Methode erlaubt zum Beispiel *Zeichenketten* (ein anderes Wort ist *Strings*) als Argumente, die dann auf der Konsole erscheinen. Ein String ist eine Folge von Buchstaben, Ziffern oder Sonderzeichen in doppelten Anführungszeichen.

Implementieren[55](»Implementieren« stammt vom lateinischen Wort »implere« ab, was für »erfüllen« und »ergänzen« steht.) wir damit eine vollständige Java-Klasse mit einem Methodenaufruf, die über `println()` etwas auf dem Bildschirm ausgibt:

Listing 2.2: Application.java

```
class Application
{
    public static void main( String[] args )
    {
        // Start des Programms

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

        // Ende des Programms
    }
}
```

3.2.5 Die API-Dokumentation

Die wichtigste Informationsquelle für Programmierer ist die offizielle API-Dokumentation von Oracle. Zu der Methode `println()` können wir bei der Klasse `PrintStream` zum Beispiel erfahren, dass diese eine Ganzzahl, eine Fließkommazahl, einen Wahrheitswert, ein Zeichen oder aber eine Zeichenkette akzeptiert. Die Dokumentation ist weder Teil vom JRE noch vom JDK – dafür ist die Hilfe zu groß. Wer über eine permanente Internetverbindung verfügt, kann die Dokumentation online unter <http://tutego.de/go/javaapi> lesen oder sie von der Oracle-

Seite <http://www.oracle.com/technetwork/java/javase/downloads/> herunterladen und als Sammlung von HTML-Dokumenten auspacken.



Abbildung 2.1: Online-Dokumentation bei Oracle

API-Dokumentation im HTML-Help-Format *

Die Oracle-Dokumentation als Loseblattsammlung hat einen Nachteil, der sich im Programmieralltag bemerkbar macht: Sie lässt sich nur ungenügend durchsuchen. Da die Webseiten statisch sind, können wir nicht einfach nach Methoden forschen, die zum Beispiel auf »listener« enden. Franck Allimant (<http://tutego.de/go/allimant>) übersetzt regelmäßig die HTML-Dokumentation von Oracle in das Format *Windows HTML-Help* (CHM-Dateien), das auch unter Unix und Mac OS X mit der Open-Source-Software <http://xchm.sourceforge.net/> gelesen werden kann. Neben den komprimierten Hilfe-Dateien lassen sich auch die Sprach- und JVM-Spezifikation sowie die API-Dokumentation der Enterprise Edition und der Servlets im Speziellen beziehen.

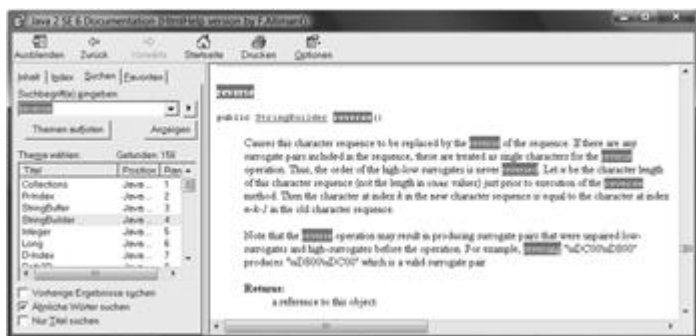


Abbildung 2.3: API-Dokumentation in der Windows-Hilfe

3.2.6 Ausdrücke

Ein *Ausdruck* (engl. *expression*) ergibt bei der Auswertung ein Ergebnis. Im Beispiel `OverloadedPrintln.java` steht in der `main()`-Methode:

```
System.out.println( "Verhaften Sie die üblichen Verdächtigen!" );
System.out.println( true );
System.out.println( -273 );
System.out.println( 1.6180339887498948 );
```

Die Argumente für `println()`, wie der String, der Wahrheitswert oder die Zahlen, sind Ausdrücke. Im dem Beispiel kommt der Ausdruck von einem Literal, aber mit Operatoren lassen sich auch komplexere Ausdrücke wie $(1 + 2) * 1.19$ bilden:

```
System.out.println( (1 + 2) * 1.19 );
```

Der Wert eines Ausdrucks wird auch *Resultat* genannt. Ausdrücke haben immer einen Wert, während das für Anweisungen (wie eine Schleife) nicht gilt. Daher kann ein Ausdruck an allen Stellen stehen, an denen ein Wert benötigt wird, etwa als Argument von `println()`. Dieser Wert ist entweder ein numerischer Wert (von arithmetischen Ausdrücken), ein Wahrheitswert (`boolean`) oder eine Referenz (etwa von einer Objekt-Erzeugung).

3.2.7 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 Radio: Ein Radio spielt Musik ab, wenn der Einschalter 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, zum Beispiel die Lautstärke 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 folgt später). Der Aufruf einer Methode auf einem Objekt richtet die Anfrage genau an ein bestimmtes Objekt. Steht in einem Java-Programm nur die Anweisung `lauter`, so weiß der Compiler nicht, wen er fragen soll, wenn es etwa drei Radio-Objekte gibt. Was ist, wenn es auch einen Fernseher mit der gleichen Operation gibt? Aus diesem Grund verbinden wir das Objekt, das etwas kann, mit der Operation. Ein Punkt trennt das Objekt von der Operation oder dem Zustand. So gehört `println()` zu einem Objekt `out`, das die Bildschirmausgabe übernimmt. Dieses Objekt `out` wiederum gehört zu der Klasse `System`.

System.out und System.err

Das Laufzeitsystem bietet uns zwei Ausgabekanäle: einen für normale Ausgaben und einen, in den wir Fehler leiten können. Der Vorteil ist, dass über diese Unterteilung die Fehler von der herkömmlichen Ausgabe getrennt werden können. Standardausgaben wandern in `System.out`, und Fehlerausgaben werden in `System.err` weitergeleitet. `out` und `err` sind vom gleichen Typ, sodass die `printXXX()`-Methoden bei beiden gleich sind:

```
System.out.println( "Das ist eine normale Ausgabe" );
System.err.println( "Das ist eine Fehlerausgabe" );
```

Die Objektorientierung wird hierbei noch einmal besonders deutlich. Das `out`- und das `err`-Objekt sind zwei Objekte, die das Gleiche können, nämlich mit `println()` etwas ausgeben. Doch ist es nicht möglich, ohne explizite Objektangabe die Methode `println()` in den Raum zu rufen und von der Laufzeitumgebung zu erwarten, dass diese weiß, ob die Anfrage an `System.out` oder an `System.err` geht.

3.2.8 Modifizierer

Die Deklaration einer Klasse oder Methode kann einen oder mehrere *Modifizierer* (engl. *modifier*) enthalten, die zum Beispiel die Nutzung einschränken oder parallelen Zugriff synchronisieren.

Beispiel

Im folgenden Programm kommen drei Modifizierer vor, die fett und unterstrichen sind:

```
public class Application
{
    public static void main( String[] args )
    {
        System.out.println( "Hallo Welt" );
    }
}
```

Der Modifizierer `public` ist ein *Sichtbarkeitsmodifizierer*. Er bestimmt, ob die Klasse beziehungsweise die Methode für Programmcode anderer Klassen sichtbar ist oder nicht. Der Modifizierer `static` zwingt den Programmierer nicht dazu, vor dem Methodenaufruf ein Objekt der Klasse zu bilden. Anders gesagt: Dieser Modifizierer bestimmt die Eigenschaft, ob sich eine Methode nur über ein konkretes Objekt aufrufen lässt oder eine Eigenschaft der Klasse ist, sodass für den Aufruf kein Objekt der Klasse nötig wird. Wir arbeiten in den ersten beiden Kapiteln nur mit statischen Methoden und werden ab Kapitel 3, »Klassen und Objekte«, nicht-statische Methoden einführen.

3.3 Datentypen, Typisierung, Variablen und Zuweisungen

Java nutzt, wie es für imperative Programmiersprachen typisch ist, Variablen zum Ablegen von Daten. Eine Variable ist ein reservierter Speicherbereich und belegt – abhängig vom Inhalt – 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, enthält. Beispiele für einfache Datentypen sind: Ganzzahlen, Fließkommazahlen, Wahrheitswerte und Zeichen. Der Typ bestimmt auch die zulässigen Operationen, denn Wahrheitswerte lassen sich nicht addieren, Ganzzahlen schon. Dagegen lassen sich Fließkommazahlen addieren, aber nicht Xor-verknüpfen. Da jede Variable einen vom Programmierer vorgegebenen festen Datentyp hat, der zur Übersetzungszeit bekannt ist und sich später nicht mehr ändern lässt, und Java stark darauf achtet, welche Operationen erlaubt sind, und auch von jedem Ausdruck spätestens zur Laufzeit den Typ kennt, ist Java eine *statisch typisierte* und *streng (stark) typisierte* Programmiersprache.[58](Während in der Literatur bei den Begriffen statisch getypt und dynamisch getypt mehr oder weniger Einigkeit herrscht, haben verschiedene Autoren unterschiedliche Vorstellungen von den Begriffen streng (stark) typisiert und schwach typisiert.)

Primitiv- oder Verweis-Typ

Die Datentypen in Java zerfallen in zwei Kategorien:

- *Primitive Typen*: Die primitiven (einfachen) Typen sind die eingebauten Datentypen für Zahlen, Unicode-Zeichen und Wahrheitswerte.
- *Referenztypen*: Mit diesem Datentyp lassen sich Objektverweise etwa auf Zeichenketten, Dialoge oder Datenstrukturen verwalten.

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

- Zu der Zeit, als Java eingeführt wurde, kannten viele Programmierer die Syntax und Semantik von C(++) und ähnlichen imperativen Programmiersprachen. Zur neuen Sprache Java zu

wechseln, fiel dadurch leichter, und es half, sich sofort auf der Insel zurechtzufinden. Es gibt aber auch Programmiersprachen wie Smalltalk, die keine primitiven Datentypen besitzen.

- Der andere Grund ist die Tatsache, dass häufig vorkommende elementare Rechenoperationen schnell durchgeführt werden müssen und bei einem einfachen Typ leicht Optimierungen durchzuführen sind.

Wir werden uns im Folgenden erst mit primitiven Datentypen beschäftigen. Referenzen werden nur dann eingesetzt, wenn Objekte ins Spiel kommen. Die nehmen wir uns in Kapitel 3, »Klassen und Objekte«, vor.

3.3.1 Primitive Datentypen im Überblick

In Java gibt es zwei Arten eingebauter Datentypen:

- *arithmetische Typen* (ganze Zahlen – auch integrale Typen genannt –, Fließkommazahlen, Unicode-Zeichen)
- *Wahrheitswerte* für die Zustände wahr und falsch

Die folgende Tabelle vermittelt dazu einen Überblick. Anschließend betrachten wir jeden Datentyp präziser.

Tabelle 2.5: Java-Datentypen und ihre Wertebereiche	
Typ	Belegung (Wertebereich)
boolean	true oder false
char	16-Bit-Unicode-Zeichen (0x0000 ... 0xFFFF)
byte	-2^7 bis $2^7 - 1$ (–128 ... 127)
short	-2^{15} bis $2^{15} - 1$ (–32.768 ... 32.767)
int	-2^{31} bis $2^{31} - 1$ (–2.147.483.648 ... 2.147.483.647)
long	-2^{63} bis $2^{63} - 1$ (–9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807)
float	1,40239846E-45f ... 3,40282347E+38f
double	4,94065645841246544E-324 ... 1,79769131486231570E+308

Bei den Ganzzahlen fällt auf, dass es eine positive Zahl »weniger« gibt als negative.

Für float und double ist das Vorzeichen nicht angegeben, da die kleinsten und größten darstellbaren Zahlen sowohl positiv wie auch negativ sein können. Mit anderen Worten: Die Wertebereiche unterscheiden sich nicht – anders als etwa bei int – in Abhängigkeit vom Vorzeichen. Wer eine »klassische« Darstellung wünscht, der kann sich das so vorstellen: Der Wertebereich (vom double) ist 4,94065645841246544E-324 bis 1,79769131486231570E+308 bzw. mit dem Vorzeichen von etwa – 1.8E308 (über –4,9E-324 und +4,9E-324) bis +1.8E308.[59](Es gibt bei Fließkommazahlen noch »Sonderzahlen«, wie plus oder minus Unendlich, aber dazu später mehr.)



Die folgende Tabelle zeigt eine etwas andere Darstellung:

Tabelle 2.6: Java-Datentypen und ihre Größen und Formate

Typ	Größe	Format
Ganzzahlen		
byte	8 Bit	Zweierkomplement
short	16 Bit	Zweierkomplement
int	32 Bit	Zweierkomplement
long	64 Bit	Zweierkomplement
Fließkommazahlen		
float	32 Bit	IEEE 754
double	64 Bit	IEEE 754
Weitere Datentypen		
boolean	1 Bit	true, false
char	16 Bit	16-Bit-Unicode

Hinweis

Strings werden bevorzugt behandelt, sind aber lediglich Verweise auf Objekte und kein primitiver Datentyp.

3.3.2 Variablendeklarationen

Mit Variablen lassen sich Daten speichern, die vom Programm gelesen und geschrieben werden können. Um Variablen zu nutzen, müssen sie deklariert (definiert[60])(In C(++) bedeuten Definition und

Deklaration etwas Verschiedenes. In Java kennen wir diesen Unterschied nicht und betrachten daher beide Begriffe als gleichwertig. Die Spezifikation spricht nur von Deklarationen.)) werden. Die Schreibweise einer Variablendeklaration ist immer die gleiche: Hinter dem Typnamen folgt der Name der Variablen. Sie ist eine Anweisung und wird daher mit einem Semikolon abgeschlossen. In Java kennt der Compiler von jeder Variablen und jedem Ausdruck genau den Typ.

Deklarieren wir ein paar (lokale) Variablen in der `main()`-Methode:

Listing 2.5: FirstVariable.java

```
public class FirstVariable
{
    public static void main( String[] args )
    {
        String name;           // Name
        int age;                // Alter
        double income;          // Einkommen
        char gender;            // Geschlecht ('f' oder 'm')
        boolean isPresident;     // Ist Präsident (true oder false)
        boolean isVegetarian;    // Ist die Person Vegetarier?
    }
}
```

Variablendeklaration mit Wertinitialisierung

Gleich bei der Deklaration lassen sich Variablen mit einem Anfangswert initialisieren. Hinter einem Gleichheitszeichen steht der Wert, der oft ein Literal ist. Ein Beispielprogramm:

Listing 2.6: Obama.java

```
public class Obama
{
    public static void main( String[] args )
    {
        String name = "Barack Hussein Obama II";
        int age = 48;
        double income = 400000;
        char gender = 'm';
        boolean isPresident = true;
    }
}
```

Zinsen berechnen als Beispiel zur Variablendeklaration, -initialisierung & -ausgabe

Zusammen mit der Konsoleneingabe können wir schon einen einfachen Zinsrechner programmieren. Er soll uns ausgeben, wie hoch die Zinsen für ein gegebenes Kapital bei einem gegebenen Zinssatz (engl. interest rate) nach einer gewissen Zeit sind.

Listing 2.7: InterestRates.java

```
public class InterestRates
{
    public static void main( String[] args )
    {
        double capital      = 20000 /* Euro */;
        double interestRate = 3.6 /* Prozent */;
        double years         = 2;



        double interestRates = capital * interestRate * years / 100;
        System.out.printf( "Zinsen: " + interestRates );    // 1440.0
    }
}
```

Das obige Beispiel macht ebenfalls deutlich, dass Strings mit dem Plus aneinandergehängt werden können; ist ein Teil kein String, so wird er in einen String konvertiert.

3.3.3 Initialisierung von lokalen Variablen

Die Laufzeitumgebung – beziehungsweise der Compiler – initialisiert lokale Variablen nicht automatisch mit einem Nullwert bzw. Wahrheitsvarianten nicht mit `false`. Vor dem Lesen müssen lokale Variablen von Hand initialisiert werden, andernfalls gibt der Compiler eine Fehlermeldung aus.[66](Anders ist das bei Objektvariablen (und statischen Variablen sowie Feldern). Sie sind standardmäßig mit `null` (Referenzen), `0` (bei Zahlen) oder `false` belegt.)

Im folgenden Beispiel seien die beiden lokalen Variablen `age` und `adult` nicht automatisch initialisiert, und so kommt es bei der versuchten Ausgabe von `age` zu einem Compilerfehler. Der Grund ist, dass ein Lesezugriff nötig ist, aber vorher noch kein Schreibzugriff stattfand.

```
int    age;
boolean adult;
System.out.println( age );    //  Local variable age may not have been
initialized.
age = 18;
if ( age >= 18 )                // Fallunterscheidung: wenn-dann
    adult = true;
System.out.println( adult );    //  Local variable adult may not have been
initialized.
```

Weil Zuweisungen in bedingten Anweisungen vielleicht nicht ausgeführt werden, meldet der Compiler auch bei `System.out.println(adult)` einen Fehler, da er analysiert, dass es einen Programmfluss ohne die Zuweisung gibt. Da `adult` nur nach der `if`-Abfrage auf den Wert `true` gesetzt wird, wäre nur unter der Bedingung, dass `age` größer gleich `18` ist, ein Schreibzugriff auf `adult` erfolgt und ein folgender Lesezugriff möglich. Doch da der Compiler annimmt, dass es andere Fälle geben kann, wäre ein Zugriff auf eine nicht initialisierte Variable ein Fehler.

3.4 Ausdrücke, Operanden und Operatoren

Beginnen wir mit mathematischen Ausdrücken, um dann die Schreibweise in Java zu ermitteln. Eine mathematische Formel, etwa der Ausdruck $-27 * 9$, besteht aus *Operanden* (engl. *operands*)

und *Operatoren* (engl. *operators*). Ein Operand ist eine Variable oder ein Literal. Im Fall einer Variablen wird der Wert aus der Variablen ausgelesen und mit ihm die Berechnung durchgeführt.

Die Arten von Operatoren

Operatoren verknüpfen die Operanden. Je nach Anzahl der Operanden unterscheiden wir:

- Ist ein Operator auf genau einem Operanden 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 Geteilt sind *binäre* (zweistellige) Operatoren.
- Es gibt auch einen Fragezeichen-Operator für bedingte Ausdrücke, der dreistellig ist.

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

3.4.1 Zuweisungsoperator

In Java dient das Gleichheitszeichen `=` der *Zuweisung* (engl. *assignment*).^[67] (Die Zuweisungen sehen zwar so aus wie mathematische Gleichungen, doch existiert ein wichtiger Unterschied: Die Formel $a = a + 1$ ist – zumindest im Dezimalsystem ohne zusätzliche Algebra – mathematisch nicht zu erfüllen, da es kein a geben kann, das $a = a + 1$ erfüllt. Aus Programmiersicht ist es in Ordnung, da die Variable a um eins erhöht wird.) Der Zuweisungsoperator ist ein binärer Operator, bei dem auf der linken Seite die zu belegende Variable steht und auf der rechten Seite ein Ausdruck.

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 ausgelesen und in den Ausdruck eingesetzt.

Mehrere Zuweisungen in einem Schritt

Zuweisungen der Form `a = b = c = 0;` sind erlaubt und gleichbedeutend mit den drei Anweisungen `c = 0; b = c; a = b;`. Die explizite Klammerung `a = (b = (c = 0))` macht noch einmal deutlich, dass sich Zuweisungen verschachteln lassen und Zuweisungen wie `c = 0` Ausdrücke sind, die einen Wert liefern. Doch auch dann, wenn wir meinen, dass

```
a = (b = c + d) + e;
```

eine coole Vereinfachung im Vergleich zu

```
b = c + d;  
a = b + e;
```

ist, sollten wir mit einer Zuweisung pro Zeile auskommen.

Die Reihenfolge der Auswertung zeigt anschaulich folgendes Beispiel:

```
int b = 10;
System.out.println( (b = 20) * b );    // 400
System.out.println( b );               // 20
```

3.4.2 Arithmetische Operatoren

Ein arithmetischer Operator verknüpft die Operanden mit den Operatoren Addition (+), Subtraktion (−), Multiplikation (*) und Division (/). Zusätzlich gibt es den Restwert-Operator (%), der den bei der Division verbleibenden Rest betrachtet. Alle Operatoren sind 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.

Numerische Umwandlung

Bei Ausdrücken mit unterschiedlichen numerischen Datentypen, etwa int und double, bringt der Compiler vor der Anwendung der Operation alle Operanden auf den umfassenderen Typ. Vor der Auswertung von $1 + 2.0$ wird somit die Ganzzahl 1 in ein double konvertiert und dann die Addition vorgenommen – das Ergebnis ist auch vom Typ double. Das nennt sich *numerische Umwandlung* (engl. *numeric promotion*). Bei byte und short gilt die Sonderregelung, dass sie vorher in int konvertiert werden.^[69](http://java.sun.com/docs/books/jls/third_edition/html/conversions.html#26917) (Auch im Java-Bytecode gibt es keine arithmetischen Operationen auf byte, short und char.) 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 zu null hin gerundet, und das Ergebnis ist keine Fließkommazahl, sodass $1/3$ das Ergebnis 0 ergibt und nicht 0,333... Den Datentyp des Ergebnisses bestimmen die Operanden und nicht der Operator. Soll das Ergebnis vom Typ double sein, muss ein Operand ebenfalls double sein.

```
System.out.println( 1.0 / 3 );          // 0.3333333333333333
System.out.println( 1   / 3.0 );        // 0.3333333333333333
System.out.println( 1   / 3 );          // 0
```

Strafe bei Division durch null

Schon die Schulmathematik lehrte uns, dass die Division durch null nicht erlaubt ist. Führen wir in Java eine Ganzzahldivision mit dem Divisor 0 durch, so bestraft uns Java mit einer ArithmeticException, die, wenn sie nicht behandelt würde, zum Ende des Programmablaufs führt. Bei Fließkommazahlen liefert eine Division durch 0 keine Ausnahme, sondern +/− unendlich und bei 0.0/0.0 den Sonderwert NaN (mehr dazu folgt in Kapitel 18, »Bits und Bytes und Mathematisches«). Ein NaN steht für *Not a Number* (auch schon manchmal »Unzahl« genannt) und wird vom Prozessor erzeugt, falls er eine mathematische Operation wie die Division durch null nicht durchführen kann. In Kapitel 12 werden wir auf NaN noch einmal zurückkommen.

Anekdote

Auf dem Lenkraketenkreuzer USS Yorktown gab ein Mannschaftsmitglied aus Versehen die Zahl Null ein. Das führte zu einer Division durch null, und der Fehler pflanzte sich so weit fort, dass die Software abstürzte und das Antriebssystem stoppte. Das Schiff trieb mehrere Stunden antriebslos im Wasser.

Der Restwert-Operator %

Eine Ganzzahldivision muss nicht unbedingt glatt aufgehen, wie im Fall von $9/2$. In diesem Fall gibt es den Rest 1. Diesen Rest liefert der *Restwert-Operator* (engl. *remainder operator*), oft auch *Modulo* genannt. Mathematiker unterscheiden die beiden Begriffe *Rest* und *Modulo*, da ein Modulo nicht negativ ist, der Rest in Java aber schon. Das soll uns aber egal sein.

```
System.out.println( 9 % 2 );           // 1
```

Der Restwert-Operator ist auch auf Fließkommazahlen anwendbar, und die Operanden können negativ sein.

```
System.out.println( 12.0 % 2.5 );      // 2.0
```

Die Division und der Restwert richten sich in Java nach einer einfachen Formel:

$$(int)(a/b) \cdot b + (a \% b) = a$$

Beispiel

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

Aus dieser Gleichung folgt, dass beim Restwert 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 Restwert-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 und bei Fließkommazahlen zum Ergebnis `NaN` führt.

Listing 2.10: RemainderAndDivDemo.java, main()

```
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
```

Gewöhnungsbedürftig ist die Tatsache, dass der erste Operand (Dividend) das Vorzeichen des Restes definiert und niemals der zweite (Divisor).

Hinweis

Um mit `value % 2 == 1` zu testen, ob `value` eine ungerade Zahl ist, muss `value` positiv sein, denn `-3 % 2` wertet Java zu `-1` aus. Der Test auf ungerade Zahlen wird erst wieder korrekt mit `value % 2 != 0`.

Restwert für Fließkommazahlen und `Math.IEEEremainder()` *

Über die oben genannte Formel können wir auch bei Fließkommazahlen das Ergebnis einer Restwert-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 Restwert-Operation den Rest von einer rundenden Division berechnet und nicht von einer abschneidenden. So wäre das Verhalten nicht analog zum Restwert bei Ganzzahlen. Java definiert den Restwert jedoch bei Fließkommazahlen genauso wie den Restwert bei Ganzzahlen. Wünschen wir ein Restwert-Verhalten, wie IEEE 754 es vorschreibt, so können wir immer noch die statische Bibliotheksmethode `Math.IEEEremainder()` [70] (Es gibt auch Methoden, die nicht mit Kleinbuchstaben beginnen, wobei das sehr selten ist und nur in Sonderfällen auftritt. `ieeeRemainder()` sah für die Autoren nicht nett aus.) verwenden.

Auch bei der Restwert-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 vorkommen, aber nicht geprüft werden.

3.4.3 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 Wert ein positiver.

Beispiel

Drehe das Vorzeichen einer Zahl um:

```
a = -a;
```

Alternativ ist:

```
a = -1 * a;
```

3.4.4 Die relationalen Operatoren und die Gleichheitsoperatoren

Relationale Operatoren sind *Vergleichsoperatoren*, die Ausdrücke miteinander vergleichen und einen Wahrheitswert vom Typ `boolean` ergeben. Die von Java für numerische Vergleiche zur Verfügung gestellten Operatoren sind:

- Größer (`>`)
- Kleiner (`<`)
- Größer-gleich (`>=`)
- Kleiner-gleich (`<=`)

Weiterhin gibt es einen Spezial-Operator `instanceof` zum Testen von Referenzeigenschaften.

Zudem kommen zwei Vergleichsoperatoren hinzu, die Java als *Gleichheitsoperatoren* bezeichnet:

- Test auf Gleichheit (==)
- Test auf Ungleichheit (!=)

Dass Java hier einen Unterschied zwischen Gleichheitsoperatoren und Vergleichsoperatoren macht, liegt an einem etwas anderen Vorrang, der uns aber nicht weiter beschäftigen soll.

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.

3.4.5 Logische Operatoren: Nicht, Und, Oder, Xor

Die Abarbeitung von Programmcode ist oft an Bedingungen geknüpft. Diese Bedingungen sind oftmals komplex zusammengesetzt, wobei drei Operatoren am häufigsten vorkommen:

- *Nicht (Negation)*: Dreht die Aussage um: Aus *wahr* wird *falsch*, und aus *falsch* wird *wahr*.
- *Und (Konjunktion)*: Beide Aussagen müssen wahr sein, damit die Gesamtaussage wahr wird.
- *Oder (Disjunktion)*: Eine der beiden Aussagen muss wahr sein, damit die Gesamtaussage wahr wird.

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 *Nicht* (!), *Und* (&&), *Oder* (||) und *Xor* (^) an. Xor ist eine Operation, die genau dann wahr liefert, wenn genau einer der beiden Operanden wahr ist. Sind beide Operanden gleich (also entweder true oder false), so ist das Ergebnis false. Xor heißt auch *exklusives beziehungsweise ausschließendes Oder*. Im Deutschen trifft die Formulierung »entweder ... oder« diesen Sachverhalt gut: Entweder ist es das eine oder das andere, aber nicht beides zusammen. Beispiel: »Willst du entweder ins Kino oder DVD schauen?«

Tabelle 2.10: Verknüpfungen der logischen Operatoren »Nicht«, »Und«, »Oder« und »Xor«

boolean a	boolean b	! a	a && b	a b	a ^ b
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	False	false

3.5 Bedingte Anweisungen oder Fallunterscheidungen

Kontrollstrukturen dienen in einer Programmiersprache dazu, Programmteile unter bestimmten Bedingungen auszuführen. Java bietet zum Ausführen verschiedener Programmteile eine if- und if-else-Anweisung sowie die switch-Anweisung. Neben der Verzweigung dienen Schleifen dazu, Programmteile mehrmals auszuführen. Bedeutend im Wort »Kontrollstrukturen« ist der Teil »Struktur«, denn die Struktur zeigt sich schon durch das bloße Hinsehen. Als es noch keine Schleifen und »hochwertigen« Kontrollstrukturen gab, sondern nur ein Wenn/Dann und einen Sprung, war die

Logik des Programms nicht offensichtlich; das Resultat nannte sich *Spaghetti-Code*. Obwohl ein allgemeiner Sprung in Java mit goto nicht möglich ist, besitzt die Sprache dennoch eine spezielle Sprungvariante. In Schleifen erlauben continue und break definierte Sprungziele.

3.5.1 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.

Mit der if-Anweisung wollen wir testen, ob der Anwender eine Zufallszahl richtig geraten hat.

Listing 2.15: WhatsYourNumber.java

```
public class WhatsYourNumber
{
    public static void main( String[] args )
    {
        int number = (int) (Math.random() * 5 + 1);

        System.out.println( "Welche Zahl denke ich mir zwischen 1 und 5?" );
        int guess = new java.util.Scanner( System.in ).nextInt();

        if ( number == guess ) {
            System.out.println( "Super getippt!" );
        }

        System.out.println( "Starte das Programm noch einmal und rate erneut!"
    );
    }
}
```

Die Abarbeitung der Ausgabe-Anweisungen hängt vom Ausdruck im if ab.

- Ist das Ergebnis des Ausdrucks wahr (number == guess wird zu true ausgewertet), wird die folgende Anweisung, also die Konsolenausgabe "Super getippt!" ausgeführt.
- Ist das Ergebnis des Ausdrucks falsch (number == guess wird zu false ausgewertet), so wird die Anweisung übersprungen, und es wird mit der ersten Anweisung nach der if-Anweisung fortgefahren.

Zusammengesetzte Bedingungen

Die bisherigen Abfragen waren sehr einfach, doch kommen in der Praxis viel komplexere Bedingungen vor. Oft im Einsatz sind die logischen Operatoren &&(Und), || (Oder), !(Nicht).

Wenn wir etwa testen wollen, ob

- eine geratene Zahl number entweder gleich der Zufallszahl guess ist oder
- eine gewisse Anzahl von Versuchen schon überschritten ist (trials größer 10),

dann schreiben wir die zusammengesetzte Bedingung so:

```
if ( number == guess || trials > 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.

3.5.2 Die Alternative mit einer if-else-Anweisung wählen

Neben der einseitigen Alternative existiert die zweiseitige Alternative. Das optionale Schlüsselwort `else` mit angehängter Anweisung veranlasst die Ausführung einer Alternative, wenn der `if`-Test falsch ist.

Rät der Benutzer aus unserem kleinen Spiel die Zahl nicht, wollen wir ihm die Zufallszahl präsentieren:

Listing 2.16: GuessTheNumber.java

```
public class GuessTheNumber
{
    public static void main( String[] args )
    {
        int number = (int) (Math.random() * 5 + 1);

        System.out.println( "Welche Zahl denke ich mir zwischen 1 und 5?" );
        int guess = new java.util.Scanner( System.in ).nextInt();

        if ( number == guess )
            System.out.println( "Super getippt!" );
        else
            System.out.printf( "Stimmt nicht, habe mir %s gedacht!", number );
    }
}
```

Falls der Ausdruck `number == guess` wahr ist, wird die erste Anweisung ausgeführt, andernfalls die zweite Anweisung. Somit ist sichergestellt, dass in jedem Fall eine Anweisung ausgeführt wird.

3.6 Schleifen

Schleifen dienen dazu, bestimmte Anweisungen immer wieder abzuarbeiten. Zu einer Schleife gehören die Schleifenbedingung und der Rumpf. Die Schleifenbedingung, ein boolescher Ausdruck, entscheidet darüber, unter welcher Bedingung die Wiederholung ausgeführt wird. 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`). Java bietet vier Typen von Schleifen:

- while-Schleife
- do-while-Schleife
- einfache for-Schleife
- erweiterte for-Schleife (auch *For-Each Loop* genannt)

Die ersten drei Schleifentypen erklären die folgenden Abschnitte, während die erweiterte for-Schleife nur bei Sammlungen nötig ist und daher später bei Feldern (siehe Kapitel 3, »Klassen und Objekte«) und dynamischen Datenstrukturen (siehe Kapitel 13, »Datenstrukturen und Algorithmen«) Erwähnung findet.

3.6.1 Die while-Schleife

Die while-Schleife ist eine abweisende Schleife, die vor jedem Schleifeneintritt die Schleifenbedingung prüft. Ist die Bedingung wahr, führt sie den Rumpf aus, andernfalls beendet sie die Schleife. Wie bei if muss auch bei den Schleifen der Typ der Bedingungen boolean sein.[78](Wir hatten das Thema bei if schon angesprochen: In C(++) ließe sich while (i) schreiben, was in Java while (i != 0) wäre.)

Beispiel

Zähle von 100 bis 40 in Zehnerschritten herunter:

Listing 2.21: WhileLoop.java, main()

```
int cnt = 100;
while ( cnt >= 40 )
{
    System.out.printf( "Ich erblickte das Licht der Welt " +
                      "in Form einer %d-Watt-Glühbirne.\n", cnt );
    cnt -= 10;
}
```

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.

Hinweis

Wird innerhalb des Schleifenkopfs schon alles Interessante erledigt, so muss trotzdem eine Anweisung folgen. Dies ist der passende Einsatz für die leere Anweisung »;« oder den leeren Block »{}«.

```
while ( ! new java.io.File( "c:/dump.bin" ).exists() )
;
```

Nur wenn die Datei existiert, läutet dies das Ende der Schleife ein; andernfalls folgt sofort ein neuer Existenztest. Tipp an dieser Stelle: Anstatt direkt zum nächsten Dateitest überzugehen, sollte eine kurze Verzögerung eingebaut werden.

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:

Listing 2.22: WhileTrue.java

```
public class WhileTrue
{
    public static void main( String[] args )
    {
        while ( true )
        {
            // immer wieder und immer wieder
        }
    }
}
```

3.6.2 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

Gib die Zahlen von 1 bis 10 auf dem Bildschirm aus:

Listing 2.24: ForLoop.java, main()

```
for ( int i = 1; i <= 10; i++ )           // i ist Schleifenzähler
    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 deklarieren und initialisieren. Diese Zählvariable ist dann außerhalb des Blocks nicht mehr gültig.[80](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 galt, andere interpretierten die Sprachspezifikation so, dass diese auch außerhalb gültig blieb. Die aktuelle C++-Definition schreibt nun vor, dass die Variable außerhalb des Blocks 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.) 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. Ist kein Test angegeben, so ist das Ergebnis automatisch `true`.
- *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 $i \leq 10$ 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.

Schleifenzähler

Wird die *for*-Schleife zum Durchlaufen einer Variablen genutzt, so heißt der *Schleifenzähler* entweder *Zählvariable* oder *Laufvariable*.

Wichtig sind die Initialisierung und die korrekte Abfrage am Ende. Schnell läuft die Schleife einmal zu oft durch und führt so zu falschen Ergebnissen. Die Fehler bei der Abfrage werden auch *off-by-one error* genannt, wenn zum Beispiel statt \leq der Operator $<$ steht. Dann nämlich läuft die Schleife nur bis 9. Ein anderer Name für den Schleifenfehler lautet *fencepost error* (Zaunpfahl-Fehler). Es geht um die Frage, wie viele Pfähle für einen 100 m langen Zaun nötig sind, sodass alle Pfähle einen Abstand von 10 m haben: 9, 10 oder 11?

Wann *for*- und wann *while*-Schleife?

Da sich die *while*- und die *for*-Schleife sehr ähnlich sind, ist die Frage berechtigt, 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 sollte vermieden werden.

for-Schleifen sollten immer dann benutzt werden, wenn eine Variable um eine konstante Größe erhöht wird. Tritt in der Schleife keine Schleifenvariable auf, die inkrementiert oder dekrementiert wird, sollte eine *while*-Schleife genutzt werden. Eine *do-while*-Schleife sollte dann eingesetzt werden, wenn die Abbruchbedingung erst am Ende eines Schleifendurchlaufs ausgewertet werden kann. Auch sollte die *for*-Schleife dort eingesetzt werden, wo sich alle drei Ausdrücke im Schleifenkopf auf dieselbe Variable beziehen. Vermieden werden sollten unzusammenhängende Ausdrücke im Schleifenkopf. Der schreibende Zugriff auf die Schleifenvariable im Rumpf ist eine schlechte Idee, wenn sie auch gleichzeitig im Kopf modifiziert wird – das ist schwer zu durchschauen und kann leicht zu Endlosschleifen führen.

Die *for*-Schleife ist nicht auf einen bestimmten Typ festgelegt, auch wenn *for*-Schleifen für das Hochzählen den impliziten Typ *int* suggerieren. Der Initialisierungsteil kann alles Mögliche vorbelegen, ob *int*, *double* oder eine Referenzvariable. Die Bedingung kann alles erdenkbare testen, nur das Ergebnis muss hier ein *boolean* sein.

3.7 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, also das, was es »kann«, wird in Java durch Methoden ausgedrückt.

Wir haben schon mindestens eine Methode kennengelernt: `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.

Für die Deklaration von Methoden gibt es drei Gründe:

- 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.
- Komplexe Programme werden in kleine Teilprogramme zerlegt, damit die Komplexität des Programms heruntergebrochen wird. Damit ist der Kontrollfluss leichter zu erkennen.
- Die Operationen einer Klasse, also das Angebot eines Objekts, sind ein Grund für Methodendeklarationen in einer objektorientierten Programmiersprache. Daneben gibt es aber noch weitere Gründe, die für Methoden sprechen. Sie werden im Folgenden erläutert.

3.7.1 Bestandteil einer Methode

Eine Methode setzt sich aus mehreren Bestandteilen zusammen. Dazu gehören der *Methodenkopf* (kurz *Kopf*) und der *Methodenrumpf* (kurz *Rumpf*). Der Kopf besteht aus einem *Rückgabotyp* (auch *Ergebnistyp* genannt), dem *Methodennamen* und einer optionalen *Parameterliste*.

Nehmen wir die bekannte statische `main()`-Methode:

```
public static void main( String[] args )
{
    System.out.println( "Wie siehst du denn aus? Biste gerannt?" );
}
```

Sie hat folgende Bestandteile:

- Die statische Methode liefert keine Rückgabe, daher ist der »Rückgabotyp« `void`. (An dieser Stelle sollte bemerkt werden, dass `void` in Java kein Typ ist.) `void` heißt auf Deutsch übersetzt: »frei«, »die Leere« oder »Hohlraum«.
- Der Methodenname ist `main`.
- Die Parameterliste ist `String[] args`.
- Der Rumpf besteht nur aus der Bildschirmausgabe.

Namenskonvention

Methodennamen beginnen wie Variablennamen mit Kleinbuchstaben und werden in der gemischten Groß-/Kleinschreibung verfasst. Bezeichner dürfen nicht wie Schlüsselwörter heißen.[81](Das führte bei manchen Bibliotheken (JUnit sei hier als Beispiel genannt) zu Überraschungen. In Java 1.4 etwa wurde das Schlüsselwort `assert` eingeführt, das JUnit als Methodenname wählte. Unzählige Zeilen Programmcode mussten daraufhin von `assert()` nach `assertTrue()` konvertiert werden.)

Die Signatur einer Methode

Der Methodenname und die Parameterliste bestimmen die *Signatur* einer Methode; der Rückgabewert und Ausnahmen gehören nicht dazu.[1] Um es ganz präzise zu machen: Typparameter gehörten auch zur Signatur – sie sind Bestandteil von Kapitel 9. Die Parameterliste ist durch die Anzahl, die Reihenfolge und die Typen der Parameter beschrieben. Pro Klasse darf es nur eine Methode mit derselben Signatur geben, sonst meldet der Compiler einen Fehler. Da die Methoden `void main(String[] args)` und `String main(String[] arguments)` die gleiche Signatur (`main, String[]`) besitzen – die Namen der Parameter spielen keine Rolle –, können sie nicht zusammen in einer Klasse deklariert werden (später werden wir sehen, dass Unterklassen durchaus gewisse Sonderfälle zulassen).

3.7.2 Aufruf einer Methode

Da eine Methode immer einer Klasse oder einem Objekt zugeordnet ist, muss der Eigentümer beim Aufruf angegeben werden. Im Fall von `System.out.println()` ist `println()` eine Methode vom `out`-Objekt. Wenn wir das Maximum zweier Fließkommazahlen mit `Math.max(a, b)` bilden, dann ist `max()` eine (statische) Methode 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 Methode. Der Methodenaufruf verzweigt in den Programmcode, aber der Aufrufer weiß nicht, was dort geschieht. Er betrachtet nur das Ergebnis.

Die aufgerufene Methode 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 Methode wie zum Beispiel `System.out.println()` gibt nichts als Ergebnis einer Berechnung zurück. Anders ist die statische Methode `max()`; sie liefert ein Ergebnis. Damit ergeben sich vier unterschiedliche Typen von Methoden:

Tabelle 2.13: Methoden mit Rückgabewerten und Parametern

Methode	Ohne Rückgabewert	Mit Rückgabewert
Ohne Parameter	<code>System.out.println()</code>	<code>System.currentTimeMillis()</code>
Mit Parameter	<code>System.out.println(4)</code>	<code>Math.max(12, 33)</code>

3.7.3 Methoden ohne Parameter deklarieren

Die einfachste Methode besitzt keinen Rückgabewert und keine Parameter. Der Programmcode steht in geschweiften Klammern hinter dem Kopf und bildet damit den Körper der Methode. Gibt die Methode nichts zurück, dann wird `void` vor den Methodennamen geschrieben. Falls die Methode etwas zurückgibt, wird der Typ der Rückgabe anstelle von `void` geschrieben.

Schreiben wir eine statische Methode ohne Rückgabe und Parameter, die etwas auf dem Bildschirm ausgibt:

Listing 2.30: FriendlyGreeter.java

```
class FriendlyGreeter
{
    static void greet()
    {
        System.out.println( "Guten Morgen. Oh, und falls wir uns nicht mehr" +
                             " sehen, guten Tag, guten Abend und gute Nacht!" );
    }

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

Eigene Methoden können natürlich wie Bibliotheksmethoden heißen, da sie zu unterschiedlichen Klassen gehören. Statt `greet()` hätten wir also den Namen `println()` vergeben dürfen.

Tipp

Die Vergabe eines Methodennamens ist gar nicht so einfach. Nehmen wir zum Beispiel an, wir wollen eine Methode schreiben, die eine Datei kopiert. Spontan kommen uns zwei Wörter in den Sinn, die zu einem Methodennamen verbunden werden wollen: »file« und »copy«. Doch in welcher Kombination? Soll `escopyFile()` oder `fileCopy()` heißen? Wenn dieser Konflikt entsteht, sollte das Verb die Aktion anführen, unsere Wahl also auf `copyFile()` fallen. Methodennamen sollten immer das Tätigkeitswort vorne haben und das Was, das Objekt, an zweiter Stelle.

3.7.4 Statische Methoden (Klassenmethoden)

Bisher arbeiten wir nur mit statischen Methoden (auch Klassenmethoden genannt). Das Besondere daran ist, dass die statischen Methoden nicht an einem Objekt hängen und daher immer ohne explizit erzeugtes Objekt aufgerufen werden können. Das heißt, statische Methoden gehören zu Klassen an sich und sind nicht mit speziellen Objekten verbunden. Am Aufruf unserer statischen Methode `greet()` lässt sich ablesen, dass hier kein Objekt gefordert ist, mit dem die Methode verbunden ist. Das ist möglich, denn die Methode ist als `static` deklariert, und innerhalb der Klasse lassen sich alle Methoden einfach mit ihrem Namen nutzen.

Statische Methoden müssen explizit mit dem Schlüsselwort `static` kenntlich gemacht werden. Fehlt der Modifizierer `static`, so deklarieren wir damit eine Objektmethode, die wir nur aufrufen können, wenn wir vorher ein Objekt angelegt haben. Das heben wir uns aber bis zum nächsten Kapitel, »Klassen und Objekte«, auf. Die Fehlermeldung sollte Ihnen aber keine Angst machen. Lassen wir von der `greet()`-Deklaration das `static` weg und ruft die statische `main()`-Methode wie jetzt ohne Aufbau eines Objekts die dann nicht mehr statische Methode `greet()` auf, so gibt es den Compilerfehler »Cannot make a static reference to the non-static method `greet()` from the type `FriendlyGreeter`«.

Ist die statische Methode in der gleichen Klasse wie der Aufrufer deklariert – in unserem Fall `main()` und `greet()` –, so ist der Aufruf allein mit dem Namen der Methode eindeutig. Befinden sich jedoch Methodendeklaration und Methodenaufruf in unterschiedlichen Klassen, so

muss der Aufrufer den Namen der Klasse nennen; wir haben so etwas schon einmal bei Aufrufen wie `Math.max()` gesehen.

<pre>class FriendlyGreeter { static void greet() { System.out.println("Moin!"); } }</pre>	<pre>class FriendlyGreeterCaller { public static void main(String[] args) { FriendlyGreeter.greet(); } }</pre>
---	--

3.7.5 Parameter, Argument und Wertübergabe

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

Beispiel

Werfen wir einen Blick auf die Methodendeklaration `printMax()`, die den größeren der beiden übergebenen Werte auf dem Bildschirm ausgibt.

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

Um die an Methoden übergebenen Werte anzusprechen, gibt es *formale Parameter*. Von unserer statischen Methode `printMax()` sind `a` und `b` die formalen Parameter der Parameterliste. Jeder Parameter wird durch ein Komma getrennt aufgelistet, wobei für jeden Parameter der Typ angegeben sein muss; eine Kurzform wie bei der sonst üblichen Variablendeklaration wie `double a, b` ist nicht möglich. Jede Parametervariable einer Methodendeklaration muss natürlich einen anderen Namen tragen.

Argumente (aktuelle Parameter)

Der Aufrufer der Methode muss für jeden Parameter ein Argument angeben. Die im Methodenkopf deklarierten Parameter sind letztendlich lokale Variablen im Rumpf der Methode. Beim Aufruf initialisiert die Laufzeitumgebung die lokalen Variablen mit den an die Methode übergebenen Argumenten. Rufen wir unsere parametrisierte Methode etwa mit `printMax(10, 20)` auf, so sind die Literale `10` und `20` *Argumente* (aktuelle Parameter der Methode). Beim Aufruf der Methode setzt die Laufzeitumgebung die Argumente in die lokalen Variablen, kopiert also den Wert `10` in die Parametervariable `a` und `20` in die Parametervariable `b`. Innerhalb des Methodenkörpers gibt es so Zugriff auf die von außen übergebenen Werte.

Das Ende des Blocks bedeutet automatisch das Ende für die Parametervariablen. Der Aufrufer weiß auch nicht, wie die Parametervariablen heißen. Eine Typanpassung von `int` auf `double` nimmt der Compiler in unserem Fall automatisch vor. Die Argumente müssen vom Typ her natürlich passen, und es gelten die für die Typanpassung bekannten Regeln.

3.7.6 Methoden mit Rückgaben

Statische Methoden wie `Math.max()` liefern in Abhängigkeit von den Argumenten ein Ergebnis zurück. Für den Aufrufer ist die Implementierung egal; er abstrahiert und nutzt lediglich die Methode statt eines Ausdrucks. Damit Methoden Rückgabewerte an den Aufrufer liefern können, müssen zwei Dinge gelten:

- Eine Methodendeklaration bekommt einen Rückgabotyp ungleich `void`.
- Eine `return`-Anweisung gibt einen Wert zurück.

Beispiel

Eine statische Methode bildet den Mittelwert und gibt diesen zurück:

```
static double avg( double x, double y )
{
    return (x + y) / 2;
}
```

Fehlt der Ausdruck, und ist es nur ein einfaches `return`, meldet der Compiler einen Programmfehler.

Obwohl einige Programmierer den Ausdruck gerne klammern, ist das nicht nötig. Klammern sollen lediglich komplexe Ausdrücke besser lesbar machen. Geklammerte Ausdrücke erinnern sonst nur an einen Methodenaufruf, und diese Verwechslungsmöglichkeit sollte bei Rückgabewerten nicht bestehen.

Der Rückgabewert muss an der Aufrufstelle nicht zwingend benutzt werden. Berechnet unsere Methode den Durchschnitt zweier Zahlen, ist es wohl eher ein Programmierfehler, den Rückgabewert nicht zu verwenden.

3.7.7 Rekursive Methoden

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

Auf dem Weg durch den Wald begegnet uns eine Fee (engl. fairy). Sie sagt 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! Selbst gießende Blumen, das letzte Ü-Ei in der Sammlung 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 –, sie besteht nämlich darin, den letzten Wunsch als »Nochmal drei Wünsche frei« zu formulieren.

Beispiel

Eine kleine Wunsch-Methode:

```
static void fairy()  
{  
    wish();  
    wish();  
    fairy();  
}
```

Durch den dauernden Aufruf der `fairy()`-Methode 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.

3.7.8 Die Türme von Hanoi

Die Legende der Türme von Hanoi soll erstmalig von Ed Lucas in einem Artikel in der französischen Zeitschrift »Cosmo« im Jahre 1890 veröffentlicht worden sein.[86](Wir halten uns hier an eine Überlieferung von C. H. A. Koster aus dem Buch »Top-down Programming with Elan« von Ellis Horwood (Verlag Ellis Horwood Ltd , ISBN 0139249370, 1987).) Der Legende nach standen vor langer Zeit im Tempel von Hanoi drei Säulen. Die erste 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 Porphyrt[87](Gestein vulkanischen Ursprungs. Besondere Eigenschaften von Porphyrt sind: hohe Bruchfestigkeit, hohe Beständigkeit gegen physikalisch-chemische Wirkstoffe und hohe Wälz- und Gleitreibung.). Die Scheibe mit dem größten Umfang lag unten, und alle kleiner werdenden Scheiben lagen obenauf. 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, so weit die Geschichte. Wollen wir den Algorithmus zur Umschichtung der Porphyrscheiben in Java programmieren, so ist eine rekursive Lösung recht einfach. Werfen wir einen Blick auf das folgende Programm, das die Umschichtungen über die drei Pflöcke (engl. *pegs*) vornimmt.

Listing 2.35: TowerOfHanoi.java

```
class TowerOfHanoi
{
    static void move( int n, String fromPeg, String toPeg, String usingPeg )
    {
        if ( n > 1 )
        {
            move( n - 1, fromPeg, usingPeg, toPeg );
            System.out.printf( "Bewege Scheibe %d von der %s zur %s.%n", n, fromPeg,
toPeg );
            move( n - 1, usingPeg, toPeg, fromPeg );
        }
        else
            System.out.printf( "Bewege Scheibe %d von der %s zur %s.%n", n, fromPeg,
toPeg );
    }

    public static void main( String[] args )
    {
        move( 4, "Kupfersäule", "Silbersäule", "Goldsäule" );
    }
}
```

Starten wir das Programm mit vier Scheiben, so bekommen wir folgende Ausgabe:

Bewege Scheibe 1 von der Kupfersäule zur Goldsäule.
Bewege Scheibe 2 von der Kupfersäule zur Silbersäule.
Bewege Scheibe 1 von der Goldsäule zur Silbersäule.
Bewege Scheibe 3 von der Kupfersäule zur Goldsäule.
Bewege Scheibe 1 von der Silbersäule zur Kupfersäule.
Bewege Scheibe 2 von der Silbersäule zur Goldsäule.
Bewege Scheibe 1 von der Kupfersäule zur Goldsäule.
Bewege Scheibe 4 von der Kupfersäule zur Silbersäule.
Bewege Scheibe 1 von der Goldsäule zur Silbersäule.
Bewege Scheibe 2 von der Goldsäule zur Kupfersäule.
Bewege Scheibe 1 von der Silbersäule zur Kupfersäule.
Bewege Scheibe 3 von der Goldsäule zur Silbersäule.
Bewege Scheibe 1 von der Kupfersäule zur Goldsäule.
Bewege Scheibe 2 von der Kupfersäule zur Silbersäule.
Bewege Scheibe 1 von der Goldsäule zur Silbersäule.

Schon bei vier Scheiben haben wir 15 Bewegungen. Selbst wenn unser Prozessor mit vielen Millionen Operationen pro Sekunde arbeitet, benötigt ein Computer für die Abarbeitung Tausende geologischer Erdzeitalter. An diesem Beispiel wird eines deutlich: Viele Dinge sind im Prinzip berechenbar, nur praktisch ist so ein Algorithmus nicht.