Galileo Computing

Java ist auch eine Insel

Das umfassende Handbuch

Bearbeitet von
Christian Ullenboom

überarbeitet 2011. Buch. ca. 1308 S.
ISBN 978 3 8362 1802 3
Format (B x L): 24 x 19 cm

schnell und portofrei erhältlich bei

beck-shop.de
DIE FACHBUCHHANDLUNG

Die Online-Fachbuchhandlung beck-shop.de ist spezialisiert auf Fachbücher, insbesondere Recht, Steuern und Wirtschaft. Im Sortiment finden Sie alle Medien (Bücher, Zeitschriften, CDs, eBooks, etc.) aller Verlage. Ergänzt wird das Programm durch Services wie Neuerscheinungsdienst oder Zusammenstellungen von Büchern zu Sonderpreisen. Der Shop führt mehr als 8 Millionen Produkte.

Christian Ullenboom

Java ist auch eine Insel

Das umfassende Handbuch

Galileo Press

Auf einen Blick

Inhalt

2 Imperative Sprachkonzepte

3 Klassen und Objekte

4 Der Umgang mit Zeichenketten

5 Eigene Klassen schreiben

6 Exceptions

7 Äußere.innere Klassen

8 Besondere Klassen der Java SE

9 Generics<T>

10 Architektur, Design und angewandte Objektorientierung

11 Die Klassenbibliothek

12 Einführung in die nebenläufige Programmierung

13 Einführung in Datenstrukturen und Algorithmen

14 Einführung in grafische Oberflächen

15 Einführung in Dateien und Datenströme

16 Einführung in die <XML>-Verarbeitung mit Java

17 Einführung ins Datenbankmanagement mit JDBC

18 Bits und Bytes und Mathematisches

Vorwort

»Mancher glaubt, schon darum höflich zu sein, weil er sich überhaupt noch der Worte und nicht der Fäuste bedient.«
– Friedrich Hebbel (1813–1863)

Am 23. Mai 1995 stellten auf der SunWorld in San Francisco der Chef vom damaligen Science Office von Sun Microsystems, John Gage, und Netscape-Mitbegründer Marc Andreessen die neue Programmiersprache Java und deren Integration in den Webbrowser Netscape vor. Damit begann der Siegeszug einer Sprache, die uns elegante Wege eröffnet, um plattformunabhängig zu programmieren und objektorientiert unsere Gedanken abzubilden. Die Möglichkeiten der Sprache und Bibliothek sind an sich nichts Neues, aber so gut verpackt, dass Java angenehm und flüssig zu programmieren ist. Dieses Buch beschäftigt sich in 19 Kapiteln mit Java, den Klassen, der Design-Philosophie und der objektorientierten Programmierung.

Über dieses Buch

Die Zielgruppe

Die Kapitel dieses Buchs sind für Einsteiger in die Programmiersprache Java wie auch für Fortgeschrittene konzipiert. Kenntnisse in einer strukturierten Programmiersprache wie C, Delphi oder Visual Basic und Wissen über objektorientierte Technologien sind hilfreich, weil das Buch nicht explizit auf eine Rechnerarchitektur eingeht oder auf die Frage, was Programmieren eigentlich ist. Wer also schon in einer beliebigen Sprache programmiert hat, der liegt mit diesem Buch genau richtig!

Was dieses Buch nicht ist

Dieses Buch darf nicht als Programmierbuch für Anfänger verstanden werden. Wer noch nie programmiert hat und mit dem Wort »Übersetzen« in erster Linie »Dolmetschen« verbindet, der sollte besser ein anderes Tutorial bevorzugen oder parallel lesen. Viele Bereiche aus dem Leben eines Industrieprogrammierers behandelt »die Insel« bis

zu einer allgemein verständlichen Tiefe, doch sie ersetzt nicht die *Java Language Specification* (JLS: *http://java.sun.com/docs/books/jls/*).

Die Java-Technologien sind in den letzten Jahren explodiert, sodass die anfängliche Überschaubarkeit einer starken Spezialisierung gewichen ist. Heute ist es kaum mehr möglich, alles in einem Buch zu behandeln, und das möchte ich mit der Insel auch auf keinen Fall. Ein Buch, das sich speziell mit der grafischen Oberfläche *Swing* beschäftigt, ist genauso umfangreich wie die jetzige Insel. Nicht anders verhält es sich mit den anderen Spezialthemen, wie etwa objektorientierter Analyse/Design, UML, verteilter Programmierung, Enterprise JavaBeans, Datenbankanbindung, OR-Mapping, Web-Services, dynamischen Webseiten und vielen anderen Themen. Hier muss ein Spezialbuch die Neugier befriedigen.

Die Insel trainiert die Syntax der Programmiersprache, den Umgang mit den wichtigen Standardbibliotheken, Entwicklungstools und Entwicklungsumgebungen, objektorientierte Analyse und Design, Entwurfsmuster und Programmkonventionen. Sie hilft aber weniger, am Abend bei der Party die hübschen Mädels und coolen IT-Geeks zu beeindrucken und mit nach Hause zu nehmen. Sorry.

Mein Leben und Java, oder warum es noch ein Java-Buch gibt

Meine ursprüngliche Beschäftigung mit Java hängt eng mit einer universitären Pflichtveranstaltung zusammen. In unserer Projektgruppe befassten wir uns 1997 mit einer objektorientierten Dialogspezifikation. Ein Zustandsautomat musste programmiert werden, und die Frage nach der Programmiersprache stand an. Da ich den Seminarteilnehmern Java vorstellen wollte, arbeitete ich einen Foliensatz für den Vortrag aus. Parallel zu den Folien erwartete der Professor eine Ausarbeitung in Form einer Seminararbeit. Die Beschäftigung mit Java machte mir Spaß und war etwas ganz anderes, als ich bis dahin gewohnt war. Vor Java kodierte ich rund 10 Jahre in Assembler, später dann mit den Hochsprachen Pascal und C, vorwiegend Compiler. Ich probierte aus, schrieb meine Erfahrungen auf und lernte dabei Java und die Bibliotheken kennen. Die Arbeit wuchs mit meinen Erfahrungen. Während der Projektgruppe sprach mich ein Kommilitone an, ob ich nicht Lust hätte, als Referent eine Java-Weiterbildung zu geben. Lust hatte ich – aber keine Unterlagen. So schrieb ich weiter, um für den Kurs Schulungsunterlagen zu haben. Als der Professor am Ende der Projektgruppe nach der Seminararbeit fragte, war die Vorform der Insel schon so umfangreich, dass die vorliegende Einleitung mehr oder weniger zur Seminararbeit wurde.

Das war 1997, und natürlich hätte ich mit dem Schreiben sofort aufhören können, nachdem ich die Seminararbeit abgegeben habe. Doch bis heute schule ich in Java, und das Schreiben ist eine Lernstrategie für mich. Wenn ich mich in neue Gebiete einarbeite, lese ich erst einmal auf Masse und beginne dann, Zusammenfassungen zu schreiben. Erst beim Schreiben wird mir richtig bewusst, was ich noch nicht weiß. Dieses Lernprinzip hat auch zu meinem ersten Buch über Amiga-Maschinensprachprogrammierung geführt. Doch das MC680x0-Buch kam nicht auf den Markt, denn die Verlage konnten mir nur mitteilen, dass die Zeit der Homecomputer vorbei sei.[1] Mit Java war das anders, denn hier war ich zur richtigen Zeit am richtigen Ort. Die Prognosen für Java stehen ungebrochen gut, weil der Einsatz von Java mittlerweile so gefestigt ist wie der von COBOL bei Banken und Versicherungen.

Heute sehe ich die Insel als ein sehr facettenreiches Java-Buch für die ambitionierten Entwickler an, die hinter die Kulissen schauen wollen. Der Detailgrad der Insel wird von keinem anderen (mir bekannten) deutsch- oder englischsprachigen Grundlagenbuch erreicht.[2] Die Erweiterung der Insel macht mir Spaß, auch wenn viele Themen kaum in einem normalen Java-Kurs angesprochen werden.

1 Damit habe ich eine Wette gegen Georg und Thomas verloren – sie durften bei einer großen Imbisskette so viel essen, wie sie wollten. Ich hatte später meinen Spaß, als wir mit dem Auto nach Hause fuhren und dreimal anhalten mussten.

2 Und vermutlich gibt es weltweit kein anderes IT-Fachbuch, das so viele unanständige Wörter im Text versteckt.

Software und Versionen

Als Grundlage für dieses Buch dient die *Java Platform Standard Edition* (Java SE) in der Version 7 in der Implementierung von Oracle, die *Java Development Kit* (Oracle JDK, kurz JDK) genannt wird. Das JDK besteht im Wesentlichen aus einem Compiler und einer Laufzeitumgebung (JVM) und ist für die Plattformen Windows, Linux und Solaris erhältlich. Ist das System kein Windows, Linux oder Solaris, gibt es Laufzeitumgebungen von anderen Unternehmen bzw. vom Hersteller der Plattform: Für Apples Mac OS X gibt es die Java-Laufzeitumgebung von Apple selbst (die bald Teil des OpenJDK sein wird), und IBM bietet für IBM System i (ehemals iSeries) ebenfalls eine Laufzeitumgebung. Die Einrichtung dieser Exoten wird in diesem Buch nicht besprochen.

Eine grafische Entwicklungsoberfläche (IDE) ist kein Teil des JDK. Zwar verlasse ich mich ungern auf einen Hersteller, weil die Hersteller unterschiedliche Entwicklergruppen ansprechen, doch sollen in diesem Buch die freien Entwicklungsumgebungen *Eclipse* und *NetBeans* Verwendung finden. Die Beispielprogramme lassen sich grundsätzlich mit beliebigen anderen Entwicklungsumgebungen, wie etwa IntelliJ IDEA oder Oracle JDeveloper, verarbeiten oder mit einem einfachen ASCII-Texteditor, wie Notepad (Windows) oder vi (Unix), eingeben und auf der Kommandozeile übersetzen. Diese Form der Entwicklung ist allerdings nicht mehr zeitgemäß, sodass ein grafischer Kommandozeilen-Aufsatz die Programmerstellung vereinfacht.

Welche Java-Version verwenden wir?

Seit Oracle (damals noch von Sun geführt) die Programmiersprache Java 1995 mit Version 1.0 vorgestellt hat, drehte sich die Versionsspirale bis Version 7 (was gleichbedeutend mit Versionsnummer 1.7 ist). Besonders für Java-Buch-Autoren stellt sich die Frage, auf welcher Java-Version ihr Text aufbauen muss und welche Bibliotheken es beschreiben soll. Ich habe das Problem so gelöst, dass ich immer die Möglichkeiten der neuesten Version beschreibe, was zur Drucklegung die Java SE 7 war. Für die Didaktik der objektorientierten Programmierung ist die Versionsfrage glücklicherweise unerheblich.

Da viele Unternehmen noch unter Java 5 entwickeln, wirft die breite Nutzung von Features der Java-Version 7 unter Umständen Probleme auf, denn nicht jedes Beispielprogramm aus der Insel lässt sich per Copy & Paste fehlerfrei in das eigene Projekt übertragen. Da Java 7 fundamentale neue Möglichkeiten in der Programmiersprache bietet, kann ein Java 5- oder Java 6-Compiler natürlich nicht alles übersetzen. Um das Problem zu entschärfen und um nicht viele Beispiele im Buch ungültig zu machen, werden die Bibliotheksänderungen und Sprachneuerungen von Java 7 zwar ausführlich in den einzelnen Kapiteln beschrieben, aber die Beispielprogramme in anderen Kapiteln bleiben

auf dem Sprachniveau von Java 5 bzw. Java 6.[3] So laufen mehrheitlich alle Programme unter den verbreiteten Versionen Java 5 und Java 6. Auch wenn die Sprachänderungen von Java 7 zu einer Verkürzung führen, ist es unrealistisch anzunehmen, dass jedes Unternehmen kurz nach der Herausgabe der neuen Java-Version und der 10. Auflage der Insel auf Java 7 wechselt. Es ist daher nur naheliegend, das Buch für eine breite Entwicklergemeinde auszulegen, anstatt für ein paar Wenige, die sofort mit der neusten Version arbeiten können. Erst in der nächsten Auflage, die synchron mit Java 8 folgt, werden die Beispiele überarbeitet und wenn möglich an die neuen Sprachmittel von Java 7 angepasst.

Das Buch in der Lehre einsetzen

»Die Insel« eignet sich ideal zum Selbststudium. Das erste Kapitel dient zum Warmwerden und plaudert ein wenig über dieses und jenes. Wer auf dem Rechner noch keine Entwicklungsumgebung installiert hat, der sollte zuerst das JDK von Oracle installieren.

Weil das JDK nur Kommandozeilentools installiert, sollte jeder Entwickler eine grafische IDE (*Integrated Development Environment*) installieren, da eine IDE die Entwicklung von Java-Programmen deutlich komfortabler macht. Eine IDE bietet gegenüber der rohen Kommandozeile einige Vorteile:

- Das Editieren, Kompilieren und Laufenlassen eines Java-Programms ist schnell und einfach über einen Tastendruck oder Mausklick möglich.
- Ein Editor sollte die Syntax von Java farbig hervorheben (Syntax-Highlighting).
- Eine kontextsensitive Hilfe zeigt bei Methoden die Parameter an, und gleichzeitig verweist sie auf die API-Dokumentation.

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

Zum Entwickeln von Software ist die Hilfe unerlässlich. Sie ist von der Entwicklungsumgebung in der Regel über einen Tastendruck einsehbar oder online zu finden. Unter welcher URL sie verfügbar ist, erklärt ebenfalls Kapitel 1.

Richtig los geht es mit Kapitel 2, und von da an geht es didaktisch Schritt für Schritt weiter. Wer Kenntnisse in C hat, kann Kapitel 2 überblättern. Wer schon in C++/C# objekt-

3 In Java 6 wurden keine neuen Spracheigenschaften hinzugefügt, nur eine Kleinigkeit bei der Gültigkeit der `@Override`-Annotation änderte sich.

orientiert programmiert hat, kann Kapitel 3 überfliegen und dann einsteigen. Objektorientierter Mittelpunkt des Buchs ist Kapitel 5: Es vermittelt die OO-Begriffe Klasse, Methode, Assoziation, Vererbung, dynamisches Binden... Nach Kapitel 5 ist die objektorientierte Grundausbildung abgeschlossen, und nach Kapitel 9 sind die Grundlagen von Java bekannt. Es folgen Vertiefungen in einzelne Bereiche der Java-Bibliothek.

Mit diesem Buch und einer Entwicklungsumgebung Ihres Vertrauens können Sie die ersten Programme entwickeln. Um eine neue Programmiersprache zu erlernen, reicht das Lesen aber nicht aus. Mit den Übungsaufgaben auf der DVD können Sie deshalb auch Ihre Fingerfertigkeit trainieren. Da Lösungen beigelegt sind, lassen sich die eigenen Lösungen gut mit den Musterlösungen vergleichen. Vielleicht bietet die Buchlösung noch eine interessante Lösungsidee oder Alternative an.

Persönliche Lernstrategien

Wer das Buch im Selbststudium nutzt, wird wissen wollen, was eine erfolgreiche Lernstrategie ist. Der Schlüssel zur Erkenntnis ist, wie so oft, die Lernpsychologie, die untersucht, unter welchen Lesebedingungen ein Text optimal verstanden werden kann. Die Methode, die ich vorstellen möchte, heißt PQ4R-Methode, benannt nach den Anfangsbuchstaben der Schritte, die die Methode vorgibt:

- *Vorschau (Preview)*: Zunächst sollten Sie sich einen ersten Überblick über das Kapitel verschaffen, etwa durch Blättern im Inhaltsverzeichnis und in den Seiten der einzelnen Kapitel. Schauen Sie sich die Abbildungen und Tabellen etwas länger an, da sie schon den Inhalt verraten und Lust auf den Text vermitteln.
- *Fragen (Question)*: Jedes Kapitel versucht, einen thematischen Block zu vermitteln. Vor dem Lesen sollten Sie sich überlegen, welche Fragen das Kapitel beantworten soll.
- *Lesen (Read)*: Jetzt geht's los, der Text wird durchgelesen. Wenn es nicht gerade ein geliehenes Bücherei-Buch ist, sollten Sie Passagen, die Ihnen wichtig erscheinen, mit vielen Farben hervorheben und mit Randbemerkungen versehen. Gleiches gilt für neue Begriffe. Die zuvor gestellten Fragen sollte jeder beantworten können. Sollten neue Fragen auftauchen – im Gedächtnis abspeichern!
- *Nachdenken (Reflect)*: Egal, ob motiviert oder nicht – das ist ein interessantes Ergebnis einer anderen Studie –, lernen kann jeder immer. Der Erfolg hängt nur davon ab, wie tief das Wissen verarbeitet wird (elaborierte Verarbeitung). Dazu müssen die Themen mit anderen Themen verknüpft werden. Überlegen Sie, wie die Aussagen mit den anderen Teilen zusammenpassen. Dies ist auch ein guter Zeitpunkt für praktische Übungen. Für die angegebenen Beispiele im Buch sollten Sie sich eigene Bei-

spiele überlegen. Wenn der Autor eine `if`-Abfrage am Beispiel des Alters beschreibt, wäre eine eigene Idee etwa eine `if`-Abfrage zur Hüpfballgröße.

- *Wiedergeben (Recite)*: Die zuvor gestellten Fragen sollten sich nun beantworten lassen, und zwar ohne den Text. Für mich ist das Schreiben eine gute Möglichkeit, um über mein Wissen zu reflektieren, doch sollte dies jeder auf seine Weise tun. Allemal ist es lustig, sich während des Duschens über alle Schlüsselwörter und ihre Bedeutung, den Zusammenhang zwischen abstrakten Klassen und Schnittstellen usw. klar zu werden. Ein Tipp: Lautes Erklären hilft bei vielen Arten der Problemlösung – quatschen Sie einfach mal den Toaster zu. Noch schöner ist es, mit jemandem zusammen zu lernen und sich gegenseitig die Verfahren zu erklären. Eine interessante Visualisierungstechnik ist die Mind-Map. Sie dient dazu, den Inhalt zu gliedern.
- *Rückblick (Review)*: Nun gehen Sie das Kapitel noch einmal durch und schauen, ob Sie alles ohne weitere Fragen verstanden haben. Manche »schnellen« Erklärungen haben sich vielleicht als falsch herausgestellt. Vielleicht klärt der Text auch nicht alles. Dann ist ein an mich gerichteter Hinweis (*c.ullenboom@tutego.de*) angebracht.

Fokus auf das Wesentliche

Einige Unterkapitel sind für erfahrene Programmierer oder Informatiker geschrieben. Besonders der Neuling wird an einigen Stellen den sequenziellen Pfad verlassen müssen, da spezielle Kapitel mehr Hintergrundinformationen und Vertrautheit mit Programmiersprachen erfordern. Verweise auf C(++), C# oder andere Programmiersprachen dienen aber nicht wesentlich dem Verständnis, sondern nur dem Vergleich.

Einsteiger in Java können noch nicht zwischen dem absolut notwendigen Wissen und einer interessanten Randnotiz unterscheiden. Die Insel gewichtet aus diesem Grund das Wissen auf zwei Arten. Zunächst gibt es vom Text abgesetzte Boxen, die zum Teil spezielle und fortgeschrittene Informationen bereitstellen. Des Weiteren enden einige Überschriften auf ein *, was bedeutet, dass dieser Abschnitt übersprungen werden kann, ohne dass dem Leser etwas Wesentliches für die späteren Kapitel fehlt.

Organisation der Kapitel

Kapitel 1, »Java ist auch eine Sprache«, zeigt die Besonderheiten der Sprache Java auf. Einige Vergleiche mit anderen populären objektorientierten Sprachen werden gezogen. Die Absätze sind nicht besonders technisch und beschreiben auch den historischen Ablauf der Entwicklung von Java. Das Kapitel ist nicht didaktisch aufgebaut, sodass einige Begriffe erst in den weiteren Kapiteln vertieft werden; Einsteiger sollten es querlesen.

Ebenso wird hier dargestellt, wie das Java JDK von Oracle zu beziehen und zu installieren ist, damit die ersten Programme übersetzt und gestartet werden können.

Richtig los geht es in **Kapitel 2**, »Imperative Sprachkonzepte«. Es hebt Variablen, Typen und die imperativen Sprachelemente hervor und schafft mit Anweisungen und Ausdrücken die Grundlagen für jedes Programm. Hier finden auch Fallanweisungen, die diversen Schleifentypen und Methoden ihren Platz. Das alles geht noch ohne große Objektorientierung.

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

Wichtig ist für viele Probleme auch der in **Kapitel 4** vorgestellte »Umgang mit Zeichenketten«. Die beiden notwendigen Klassen `Character` für einzelne Zeichen und `String`, `StringBuffer`/`StringBuilder` für Zeichenfolgen werden eingeführt, und auch ein Abschnitt über reguläre Ausdrücke fehlt nicht. Bei den Zeichenketten müssen Teile ausgeschnitten, erkannt und konvertiert werden. Ein `split()` vom `String` und der `Scanner` zerlegen Zeichenfolgen anhand von Trennern in Teilzeichenketten. Format-Objekte bringen beliebige Ausgaben in ein gewünschtes Format. Dazu gehört auch die Ausgabe von Dezimalzahlen.

Mit diesem Vorwissen über Objekterzeugung und Referenzen kann der nächste Schritt erfolgen: In **Kapitel 5** werden wir »Eigene Klassen schreiben«. Anhand von Spielen und Räumen modellieren wir Objekteigenschaften und zeigen Benutzt- und Vererbungsbeziehungen auf. Wichtige Konzepte – wie statische Eigenschaften, dynamisches Binden, abstrakte Klassen und Schnittstellen (Interfaces) sowie Sichtbarkeit – finden dort ihren Platz. Da Klassen in Java auch innerhalb anderer Klassen liegen können (innere Klassen), setzt sich ein eigenes Unterkapitel damit auseinander.

Ausnahmen, die wir in **Kapitel 6**, »Exceptions«, behandeln, bilden ein wichtiges Rückgrat in Programmen, da sich Fehler kaum vermeiden lassen. Da ist es besser, die Behandlung aktiv zu unterstützen und den Programmierer zu zwingen, sich um Fehler zu kümmern und diese zu behandeln.

Kapitel 7, »Äußere.innere Klassen«, beschreibt, wie sich Klassen ineinander verschachteln lassen. Das verbessert die Kapselung, denn auch Implementierungen können dann sehr lokal sein.

Kapitel 8, »Besondere Klassen der Java SE«, geht auf die Klassen ein, die für die Java-Bibliothek zentral sind, etwa Vergleichsklassen, Wrapper-Klassen oder die Klasse `Object`, die die Oberklasse aller Java-Klassen ist.

Mit Generics lassen sich Klassen, Schnittstellen und Methoden mit einer Art Typ-Platzhalter deklarieren, wobei der konkrete Typ erst später festgelegt wird. **Kapitel 9**, »Generics<T>«, gibt einen Einblick in die Technik.

Danach sind die Fundamente gelegt, und die verbleibenden Kapitel dienen dazu, das bereits erworbene Wissen auszubauen. **Kapitel 10**, »Architektur, Design und angewandte Objektorientierung«, zeigt Anwendungen guter objektorientierter Programmierung und stellt Entwurfsmuster (Design-Pattern) vor. An unterschiedlichen Beispielen demonstriert das Kapitel, wie Schnittstellen und Klassenhierarchien gewinnbringend in Java eingesetzt werden. Es ist der Schlüssel dafür, nicht nur im Kleinen zu denken, sondern auch große Applikationen zu schreiben.

Nach den ersten zehn Kapiteln haben die Leser die Sprache Java nahezu komplett kennengelernt. Da Java aber nicht nur eine Sprache ist, sondern auch ein Satz von Standardbibliotheken, konzentriert sich die zweite Hälfe des Buchs auf die grundlegenden APIs. Jeweils am Ende eines Kapitels findet sich ein Unterkapitel »Zum Weiterlesen« mit Verweisen auf interessante Internetadressen – in der Java-Sprache `finally{}` genannt. Hier kann der Leser den sequenziellen Pfad verlassen und sich einzelnen Themen widmen, da die Themen in der Regel nicht direkt voneinander abhängen.

Die Java-Bibliothek besteht aus mehr als 4.000 Klassen, Schnittstellen, Aufzählungen, Ausnahmen und Annotationen. Das **Kapitel 11**, »Die Klassenbibliothek«, (und auch der Anhang A) gibt eine Übersicht über die wichtigsten Pakete und greift einige Klassen aus der Bibliothek heraus, etwa zum Laden von Klassen. Hier sind auch Klassen zur Konfiguration von Anwendungen oder Möglichkeiten zum Ausführen externer Programme zu finden.

Die Kapitel 11 bis 16 geben einen Überblick über spezielle APIs auf. Die Bibliotheken sind sehr umfangreich, und das aufbauende Java-Expertenbuch vertieft die API weiter. **Kapitel 12** gibt eine »Einführung in die nebenläufige Programmierung«. **Kapitel 13**, »Einführung in Datenstrukturen und Algorithmen«, zeigt praxisnah geläufige Datenstrukturen wie Listen, Mengen und Assoziativspeicher. Einen Kessel Buntes bietet **Kapitel 14**, »Einführung in grafische Oberflächen«, wo es um die Swing-Bibliothek geht. Darüber, wie aus Dateien gelesen und geschrieben wird, gibt **Kapitel 15**, »Einführung in Dateien und Datenströme«, einen Überblick. Da Konfigurationen und Daten oftmals im XML-Format vorliegen, zeigt **Kapitel 16**, »Einführung in die <XML>-Verarbeitung mit Java«, auf, welche Möglichkeiten zur XML-Verarbeitung die Bibliothek bietet. Wer stattdessen eine

relationale Datenbank bevorzugt, der findet in **Kapitel 17**, »Einführung ins Datenbankmanagement mit JDBC«, Hilfestellungen. Alle genannten Kapitel geben aufgrund der Fülle nur einen Einblick, und ihre Themen werden im zweiten Band noch tiefer aufgefächert.

Kapitel 18 stellt »Bits und Bytes und Mathematisches« vor. Die Klasse `Math` hält typische mathematische Methoden bereit, um etwa trigonometrische Berechnungen durchzuführen. Mit einer weiteren Klasse können Zufallszahlen erzeugt werden. Auch behandelt das Kapitel den Umgang mit beliebig langen Ganz- oder Fließkommazahlen. Die meisten Entwickler benötigen nicht viel Mathematik, daher ist es das Schlusskapitel.

Abschließend liefert **Kapitel 19**, »Die Werkzeuge des JDK«, eine Kurzübersicht der Kommandozeilenwerkzeuge *javac* zum Übersetzen von Java-Programmen und *java* zum Starten der JVM und Ausführen der Java-Programme.

Anhang A, »Die Klassenbibliothek« erklärt alle Java-Pakete kurz mit einem Satz und zudem alle Typen im absolut essenziellen Paket `java.lang`.

Konventionen

In diesem Buch werden folgende Konventionen verwendet:

- Neu eingeführte Begriffe sind *kursiv* gesetzt, und der Index verweist genau auf diese Stelle. Des Weiteren sind *Dateinamen*, *HTTP-Adressen*, *Namen ausführbarer Programme, Programmoptionen* und *Dateiendungen* (*.txt*) kursiv. Einige Links führen nicht direkt zur Ressource, sondern werden über *http://www.tutego.de/go* zur tatsächlichen Quelle umgeleitet, was Änderungen erleichtert.
- Begriffe der Benutzeroberfläche stehen in KAPITÄLCHEN.
- Listings, Methoden und sonstige Programmelemente sind in `nicht-proportionaler` Schrift gesetzt. An einigen Stellen wurde hinter eine Listingzeile ein abgeknickter Pfeil als Sonderzeichen gesetzt, das den Zeilenumbruch markiert. Der Code aus der nächsten Zeile gehört also noch zur vorigen.
- Um im Programmcode Compilerfehler oder Laufzeitfehler anzuzeigen, steht in der Zeile ein ☠. So ist auf den ersten Blick abzulesen, dass die Zeile nicht compiliert wird oder zur Laufzeit aufgrund eines Programmierfehlers eine Ausnahme auslöst. Beispiel:

```
int p = new java.awt.Point();     // ☠ Compilerfehler: Type mismatch
```

- Bei Compilerfehlern – wie im vorangehenden Punkt – kommen die Fehlermeldungen in der Regel von Eclipse. Sie sind dort anders benannt als in NetBeans bzw. dem Kommandozeilencompiler *javac*. Aber natürlich führen beide Compiler zu ähnlichen Fehlern.
- Bei Methodennamen im Fließtext folgt immer ein Klammerpaar. Die Parameter werden nur dann aufgeführt, wenn sie wichtig sind.
- Um eine Gruppe von Methoden anzugeben, symbolisiert die Kennung `XXX` einen Platzhalter. So zeigt zum Beispiel `printXXX()` die Methoden `println()`, `print()` und `printf()` an. Aus dem Kontext geht hervor, welche Methoden gemeint sind.
- Raider heißt jetzt Twix, und Sun ging Anfang 2010 an Oracle. Auch wenn es für langjährige Entwickler hart ist: Der Name »Sun« verschwindet, und der geliebte Datenbankhersteller tritt an seine Stelle. Er taucht immer nur dann auf, wenn es um eine Technologie geht, die von Sun initiiert wurde und in der Zeit auf den Markt kam, in der Sun sie verantwortete.

Programmlistings

Komplette Programmlistings sind wie folgt aufgebaut:

Listing 0.1: Person.java

```
class Person
{
}
```

Der abgebildete Quellcode befindet sich in der Datei *Person.java*. Befindet sich der Typ (Klasse, Aufzählung, Schnittstelle, Annotation) in einem Paket, steht die Pfadangabe beim Dateinamen:

Listing 0.2: com/tutego/insel/Person.java

```
package com.tutego.insel.Person;
class Person { }
```

Um Platz zu sparen, stellt das Buch oftmals Quellcode-Ausschnitte dar. Der komplette Quellcode ist auf der DVD beziehungsweise im Internet verfügbar. Hinter dem Typ folgen in dem Fall Kennungen des abgedruckten Teils. Ist nur die Typdeklaration einer Datei ohne `package`- oder `import`-Deklaration aufgelistet, so steht hinter dem Dateinamen der Typ, etwa so:

Listing 0.3: Person.java, Person

Listing 0.4: Person.java, House

Im folgenden Fall wird nur die `main()`-Methode abgebildet:

Listing 0.5: Person.java, main()

Wird ein Ausschnitt einer Datei *Person.java* abgebildet, steht »Ausschnitt« oder »Teil 1«, »Teil 2«... dabei:

Listing 0.6: Person.java, Ausschnitt

Listing 0.7: Person.java, main() Teil 1

Gibt es Beispielprogramme für bestimmte Klassen, so enden die Klassennamen dieser Programme im Allgemeinen auf *-Demo*. Für die Java-Klasse `DateFormat` heißt somit ein Beispielprogramm, das die Funktionalität der Klasse `DateFormat` vorführt, *DateFormat-Demo*.

API-Dokumentation im Buch

Attribute, Konstruktoren und Methoden finden sich in einer speziellen Auflistung, die es ermöglicht, sie leicht im Buch zu finden und die Insel als Referenzwerk zu nutzen.

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

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

Im Rechteck steht der vollqualifizierte Klassen- oder Schnittstellenname (etwa die Klasse `DateFormat` im Paket `java.text`) beziehungsweise der Name der Annotation. In den nachfolgenden Zeilen sind die Oberklasse (`DateFormat` erbt von `Format`) und die implementierten Schnittstellen (`DateFormat` implementiert `Cloneable` und `Serializable`) aufgeführt. Da jede Klasse, die keine explizite Oberklasse hat, automatisch von `Object` erbt, ist diese nicht extra angegeben. Die Sichtbarkeit ist, wenn nicht anders angegeben, `public`, da dies für Bibliotheksmethoden üblich ist. Wird eine Schnittstelle beschrieben, sind die Methoden automatisch abstrakt und öffentlich, und die Schlüsselwörter `abstract` und `public` werden nicht zusätzlich angegeben. In der anschließenden Aufzählung folgen Konstruktoren, Methoden und Attribute. Wenn nicht anders angegeben, ist die Sichtbarkeit `public`. Sind mit `throws` Fehler angegeben, dann handelt es sich nicht um

RuntimeExceptions, sondern nur um geprüfte Ausnahmen. Veraltete (deprecated) Methoden sind nicht aufgeführt, lediglich, wenn es überhaupt keine Alternative gibt.

Ausführbare Programme

Ausführbare Programme auf der Kommandozeile sind durch ein allgemeines Dollarzeichen am Anfang zu erkennen (auch wenn andere Betriebssysteme und Kommandozeilen ein anderes Prompt anzeigen). Die vom Anwender einzugebenden Zeichen sind fett gesetzt, die Ausgabe nicht:

```
$ java FirstLuck
Hart arbeiten hat noch nie jemanden getötet. Aber warum das Risiko auf sich nehmen?
```

Über die richtige Programmierer-»Sprache«

Die Programmierer-Sprache in diesem Buch ist Englisch, um ein Vorbild für »echte« Programme zu sein. Bezeichner wie Klassennamen, Methodennamen und auch eigene API-Dokumentationen sind auf Englisch, um eine Homogenität mit der englischen Java-Bibliothek zu schaffen. Zeichenketten und Konsolenausgaben sowie die Zeichenketten in Ausnahmen (Exceptions) sind in der Regel auf Deutsch, da es in realistischen Programmen kaum hart einkodierte Meldungen gibt – spezielle Dateien halten unterschiedliche Landessprachen vor. Zeilenkommentare sind als interne Dokumentation ebenfalls auf Deutsch vorhanden.

Online-Informationen und -Aufgaben

Dieses Buch ist in der aktuellen Version im Internet unter der Adresse *http://www.tutego.de/javabuch/* und *http://www.galileocomputing.de/* erhältlich. Die Webseiten informieren umfassend über das Buch und über die kommenden Versionen, etwa Erscheinungsdatum oder Bestellnummer. Der Quellcode der Beispielprogramme ist entweder komplett oder mit den bedeutenden Ausschnitten im Buch abgebildet. Ein Zip-Archiv mit allen Beispielen ist auf der Buch-Webseite erhältlich sowie auf die Buch-DVD gepresst. Alle Programmteile sind frei von Rechten und können ungefragt in eigene Programme übernommen und modifiziert werden.

Wer eine Programmiersprache erlernen möchte, muss sie wie eine Fremdsprache sprechen. Begleitend gibt es eine Aufgabensammlung unter *http://www.tutego.de/aufgaben/j/*, die ebenfalls auf der DVD ist. Viele Musterlösungen sind dabei. Die Seite wird in regelmäßigen Abständen mit neuen Aufgaben und Lösungen aktualisiert.

Passend zur Online-Version verschließt sich das Buch nicht den Kollaborationsmöglichkeiten des Web 2.0. Neue Kapitel und Abschnitte des Buches werden immer im Java-Insel-Blog *http://javainselblog.tutego.de/* veröffentlicht.

Abbildung 1: Der Blog zum Buch und mit tagesaktuellen Java-News

Leser erfahren im Blog von allen Aktualisierungen im Buch und können das Geschehen kommentieren. Neben den reinen Updates aus dem Buch publiziert der Blog auch tagesaktuelle Nachrichten über die Java-Welt und Java-Tools. Facebook-Nutzer können ein Fan der Insel werden (*http://www.facebook.com/pages/Javainsel/157203814292515*), und Twitter-Nutzer können den Nachrichtenstrom unter *http://twitter.com/javabuch* abonnieren.

Weiterbildung durch tutego

Unternehmen, die zur effektiven Weiterbildung ihrer Mitarbeiter IT-Schulungen wünschen, können einen Blick auf *http://www.tutego.de/seminare/* werfen. tutego bietet über hundert IT-Seminare zu Java-Themen, C(++), C#/.NET, Datenbanken (Oracle, MySQL), XML (XSLT, Schema), Netzwerken, Internet, Office etc. Zu den Java-Themen zählen unter anderem:

- Java-Einführung, Java für Fortgeschrittene, Java für Umsteiger
- Softwareentwicklung mit Eclipse
- nebenläufiges Programmieren mit Threads
- JavaServer Faces (JSF), JavaServer Pages (JSP), Servlets und weitere Web-Technologien
- Datenbankanbindung mit JDBC, OR-Mapping mit JPA und Hibernate
- Java EE, EJB
- grafische Oberflächen mit Swing und JFC; Eclipse RPC, SWT
- Java und XML, JAXB
- Android

Danksagungen

Der größte Dank gebührt Sun Microsystems, die 1991 mit der Entwicklung begannen. Ohne Sun gäbe es kein Java, und ohne Java gäbe es auch nicht dieses Java-Buch. Dank gehört auch der Oracle Company als Käufer von Sun, denn vielleicht wäre ohne die Übernahme Java bald am Ende gewesen.

Die professionellen, aufheiternden Comics stammen von Andreas Schultze (*Akws@aol.com*). Ich danke auch den vielen Buch- und Artikelautoren für ihre interessanten Werke, aus denen ich mein Wissen über Java schöpfen konnte. Ich danke meinen Eltern für ihre Liebe und Geduld und meinen Freunden und Freundinnen für ihr Vertrauen. Ein weiteres Dankeschön geht an verschiedene treue Leser, deren Namen aufzulisten viel Platz kosten würde; ihnen ist die Webseite *http://www.tutego.de/javabuch/korrekteure.htm* gewidmet.

Java lebt – vielleicht sollte ich sogar »überlebt« sagen ... – durch viele freie gute Tools und eine aktive Open-Source-Community. Ein Dank geht an alle Entwickler, die großartige Java-Tools wie Eclipse, NetBeans, Ant, Maven, GlassFish, Tomcat, JBoss und Hunderte andere Bibliotheken schreiben und warten: Ohne Sie wäre Java heute nicht da, wo es ist.

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

Feedback

Auch wenn wir die Kapitel noch so sorgfältig durchgegangen sind, ist es nicht auszuschließen, dass es noch Unstimmigkeiten[4] gibt; vielmehr ist es bei 1.000 Seiten wahrscheinlich. Wer Anmerkungen, Hinweise, Korrekturen oder Fragen zu bestimmten Punkten oder zur allgemeinen Didaktik hat, der sollte sich nicht scheuen, mir eine E-Mail unter der Adresse *c.ullenboom@tutego.de* zu senden. Ich bin für Anregung, Lob und Tadel stets empfänglich.

In der Online-Version des Buchs haben wir eine besondere Möglichkeit zur Rückmeldung: Unter jedem Kapitel gibt es eine Textbox, sodass Leser uns schnell einen Hinweis schicken können. In der Online-Version können wir zudem Fehler schnell korrigieren, denn es gibt zum Teil bedauerliche Konvertierungsprobleme vom Buch ins HTML-Format, und einige Male blieb das Hochzeichen (^) auf der Strecke, sodass statt »2^16« im Text ein »216« die Leser verwunderte.

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

Sonsbeck im Jahr 2011, Jahr 1 nach Oracles Übernahme
Christian Ullenboom

Vorwort zur 10. Auflage

Die größte Neuerung in der 10. Auflage ist die Aufspaltung der Insel in ein Einführungsbuch und ein Fortgeschrittenenbuch. Eine Aufteilung wurde aus zwei Gründen nötig: a) Die Insel kam mit mehr als 1.400 Seiten an ihr druckbares Ende. Da die Java-Bibliotheken aber immer größer wurden und die Syntax (in langsamem Tempo) ebenso zunahm, mussten mehr und mehr Absätze aus der Insel ausgelagert werden. Dadurch verlor die Insel an Tiefe, eine Eigenschaft, die Leser aber an diesem Buch liebten. Der zweite Grund für ein Splitting ist, dass Spracheinsteiger, die beginnen, Variablen zu deklarieren und Klassen zu modellieren, nicht in einem Rutsch gleich mit Generics beginnen, RMI-Aufrufe starten oder mit JNI auf C-Funktionen zugreifen – für Einsteiger wäre das eine Art Bulimie-Wissen: da rein, da raus. Daher adressieren die beiden Bücher zwei unterschiedliche Zielgruppen: Dieses Buch spricht Einsteiger in Java an, die die Sprache und ihre Standardbibliothek praxisnah und in vielen Facetten lernen möchten. Fortgeschrittene Java-Entwickler mit längerer Praxiserfahrung bekommen im zweiten Buch einen tiefe-

4 Bei mir wird gerne ein »wir« zum »wie« – wie(r) dumm, dass die Tasten so eng beieinanderliegen.

ren Einblick in Generics und in die Java SE-Bibliotheken sowie einen Ausblick auf Web- und Swing-Programmierung.

Gegenüber der 9. Auflage ergeben sich folgende Änderungen: Das zweite Kapitel mit den imperativen Konzepten ist um ein Zahlenratespiel erweitert worden, sodass die Beispiele nicht so trocken, sondern anschaulicher sind. Zudem war in den Türmen von Hanoi die Silber- und Gold-Säule vertauscht, was mehrere Jahre keinem aufgefallen ist und mich dazu verleitet anzunehmen, dass die Rekursion mit den Türmen nur auf ein geringes Interesse stößt. Die Einführung in Unicode, die ebenfalls in Kapitel 2 stand, ist ins Kapitel 4 gewandert, was sich nun komplett um Zeichen und Zeichenketten kümmert; vorher war das Thema unnötig gespalten und das Kapitel am Anfang zu detaillastig.

Durch die Version 7 gibt es nur wenige Änderungen, und hier sind die Zuwächse eher minimal. Die Neuerungen in der Sprache lassen sich an einer Hand abzählen: Unterstriche in Literalen, `switch` mit String, Binär/Short-Präfixe, Diamanten-Typ, Multi-Catch bei Ausnahmen, präzisiertes Auslösen von Ausnahmen, ARM-Blöcke … Im Buch macht das vielleicht 1 % der Änderungen aus.

Vorwort zur 9. Auflage

Neben Detailverbesserungen habe ich das Generics-Kapitel komplett neu geschrieben, und viele Abschnitte und Kapitel umsortiert, um sie didaktisch leichter zugänglich zu machen. Auch sprachlich ist die Insel wieder etwas präziser geworden: Der Begriff »Funktion« für eine statische Methode ist abgesetzt, und es heißt jetzt »statische Methode« oder eben »Objektmethode«, wenn der Unterschied wichtig ist, und einfach nur »Methode«, wenn der Unterschied nicht relevant ist. Dass Java von Sun zu Oracle übergegangen ist und vollständig Open Source ist, bleibt auch nicht unerwähnt, genauso wie neue Technologien, zu denen etwa JavaFX gehört. Durch diesen erhöhten Detailgrad mussten leider einige Kapitel (wie JNI, Java ME) aus der Insel fallen. Weiterhin gibt es Bezüge zu der kommenden Version Java 7 und viele interessante Sprachvergleiche, wie Features in anderen Programmiersprachen aussehen und inwiefern sie sich von Java unterscheiden.

Nach dem Vorwort ist es jetzt jedoch an Zeit, zur Sache zu kommen und dem griechischen Philosophen Platon zu folgen, der sagte: »Der Beginn ist der wichtigste Teil der Arbeit.«

Kapitel 2

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.

2.1 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.

2.1.1 Token

Ein *Token* ist eine lexikalische Einheit, die dem Compiler die Bausteine des Programms liefert. Der Compiler erkennt an der Grammatik einer Sprache, welche Folgen von Zei-

chen ein Token bilden. Für Bezeichner heißt dies beispielsweise: »Nimm die nächsten Zeichen, solange auf einen Buchstaben nur Buchstaben oder Ziffern folgen.« Eine Zahl wie 1982 bildet zum Beispiel ein Token durch folgende Regel: »Lies so lange Ziffern, bis keine Ziffer mehr folgt.« Bei Kommentaren bilden die Kombinationen /* und */ ein Token.[1]

Whitespace

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

Folgendes ist alles andere als gut zu lesen, obwohl der Compiler es akzeptiert:

```
class _{static long _
(long __,long  ___) {
return __==0 ?___+ 1:
___==0?_(__-1,1):_(__
-1,_(__, ___-1)) ;  }
static  {int _=2 ,___
= 2;System.out.print(
"a("+_+','+___+ ")="+
_ (_,  ___) ) ;System
.exit(1);}}//(C) Ulli
```

Neben den Trennern gibt es noch 9 Zeichen, die als *Separator* definiert werden:

```
; , . ( ) { } [ ]
```

1 Das ist in C(++) unglücklich, denn so wird ein Ausdruck `*s/*t` nicht wie erwartet geparst. Erst ein Leerzeichen zwischen dem Geteiltzeichen und dem Stern »hilft« dem Parser, die gewünschte Division zu erkennen.

2.1.2 Textkodierung durch Unicode-Zeichen

Java kodiert Texte durch *Unicode-Zeichen*. Jedem Zeichen ist ein eindeutiger Zahlenwert (engl. *code point*) zugewiesen, sodass zum Beispiel das große A an Position 65 liegt. Der Unicode-Zeichensatz beinhaltet die ISO-US-ASCII-Zeichen[2] von 0 bis 127 (hexadezimal 0x00 bis 0x7f, also 7 Bit) und die erweiterte Kodierung nach ISO 8859-1 (Latin-1), die Zeichen von 128 bis 255 hinzunimmt. Mehr Details zu Unicode liefert Kapitel 4, »Der Umgang mit Zeichenketten«.

2.1.3 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.[3] 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 zB

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

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

2 *http://en.wikipedia.org/wiki/ASCII*

3 Die Java-Methoden `Character.isJavaIdentifierStart()`/`isJavaIdentiferPart()` stellen auch fest, ob Zeichen Java-Identifier sind.

zB **Beispiel** (Forts.)

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.

Aufbau der Bezeichner

Jeder Java-Bezeichner ist eine Folge aus *Java-Buchstaben* und *Java-Ziffern,*[4] wobei der Bezeichner mit einem Java-Buchstaben beginnen muss. Ein Java-Buchstabe umfasst nicht nur unsere lateinischen Buchstaben aus dem Bereich »A« bis »Z« (auch »a« bis »z«), sondern auch viele weitere Zeichen aus dem Unicode-Alphabet, etwa den Unterstrich, Währungszeichen – wie die Zeichen für Dollar ($), Euro (€), Yen (¥) – oder griechische Buchstaben. Auch wenn damit viele wilde Zeichen als Bezeichner-Buchstaben grundsätzlich möglich sind, sollte doch die Programmierung mit englischen Bezeichnernamen erfolgen. Es ist noch einmal zu betonen, dass Java streng zwischen Groß- und Kleinschreibung unterscheidet.

Die folgende Tabelle listet einige gültige Bezeichner auf:

Gültige Bezeichner	Grund
`Mami`	`Mami` besteht nur aus Alphazeichen und ist daher korrekt.
`__RAPHAEL_IST_LIEB__`	Unterstriche sind erlaubt.
`bóõlêáñ`	Ist korrekt, auch wenn es Akzente enthält.
α	Das griechische Alpha ist ein gültiger Java-Buchstabe.
`REZE$$SION`	Das Dollar-Zeichen ist ein gültiger Java-Buchstabe.
¥€$	Tatsächlich auch gültige Java-Buchstaben

Tabelle 2.1: Beispiele für gültige Bezeichner in Java

4 Ob ein Zeichen ein Buchstabe ist, stellt die statische Methode `Character.isLetter()` fest; ob er ein gültiger Bezeichner-Buchstabe ist, sagen die Funktionen `isJavaIdentifierStart()` für den Startbuchstaben und `isJavaIdentifierPart()` für den Rest.

Ungültige Bezeichner dagegen sind:

Ungültige Bezeichner	Grund
`2und2macht4`	Das erste Symbol muss ein Java-Buchstabe sein und keine Ziffer.
`hose gewaschen`	Leerzeichen sind in Bezeichnern nicht erlaubt.
`Faster!`	Das Ausrufezeichen ist, wie viele Sonderzeichen, ungültig.
`null`, `class`	Der Name ist schon von Java belegt. `Null` – Groß-/Kleinschreibung ist relevant – oder `cláss` wären möglich.

Tabelle 2.2: Beispiele für ungültige Bezeichner in Java

Hinweis

In Java-Programmen bilden sich Bezeichnernamen oft aus zusammengesetzten Wörtern einer Beschreibung. Dies bedeutet, dass in einem Satz wie »open file read only« die Leerzeichen entfernt werden und die nach dem ersten Wort folgenden Wörter mit Großbuchstaben beginnen. Damit wird aus dem Beispielsatz anschließend »openFileReadOnly«. Sprachwissenschaftler nennen einen Großbuchstaben inmitten von Wörtern *Binnenmajuskel*.

2.1.4 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 zB

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

zB **Beispiel** (Forts.)

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

2.1.5 Reservierte Schlüsselwörter

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

zB **Beispiel**

Reservierte Schlüsselwörter sind im Folgenden fett und unterstrichen gesetzt.

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

Schlüsselwörter und Literale in Java

Nachfolgende Zeichenfolgen sind Schlüsselwörter (beziehungsweise Literale im Fall von `true`, `false` und `null`)[5] und sind in Java daher nicht als Bezeichnernamen möglich.

5 Siehe dazu Abschnitt 3.9, »Keywords«, der Sprachdefinition unter *http://java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.9*.

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

Tabelle 2.3: Reservierte Schlüsselwörter in Java

Obwohl die mit † gekennzeichneten Wörter zurzeit nicht von Java benutzt werden, können doch keine Variablen dieses Namens deklariert werden.

2.1.6 Zusammenfassung der lexikalischen Analyse

Übersetzt der Compiler Java-Programme, so beginnt er mit der lexikalischen Untersuchung des Quellcodes. Wir haben dabei die zentralen Elemente schon kennengelernt, und diese sollen hier noch einmal zusammengefasst werden. Nehmen wir dazu das folgende einfache Programm:

```
class Application
{
  public static void main( String[] args )
  {
    String text = "Hallo Welt " + 21;
    System.out.println( text );
  }
}
```

Der Compiler überliest alle Kommentare, und die Trennzeichen bringen den Compiler von Token zu Token. Folgende Tokens lassen sich im Programm ausmachen:

Token-Typ	Beispiel	Erklärung
Bezeichner	`Application`, `main`, `args`, `text`, `System`, `out`, `println`	Namen für Klasse, Variable, Methode, ...
Schlüsselwort	`class`, `public`, `static`, `void`	Reservierte Wörter
Literal	`"Hallo Welt"`, `21`	Konstante Werte, wie Strings, Zahlen, ...
Operator	`=`, `+`	Operator für Zuweisungen, Berechnungen, ...
Trennzeichen	`(`, `)`, `{`, `}`, `;`	Symbole, die neben dem Trennzeichen die Tokens trennen

Tabelle 2.4: Token des Beispielprogramms

2.1.7 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[6] `//` 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.

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

- *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 */ )
  {
  }
}
```

Für den Compiler sieht die Klasse mit den Kommentaren genauso aus wie ohne, also wie `class DoYouHaveAnyCommentsToMake { }`. Im Bytecode steht exakt das Gleiche – alle Kommentare werden vom Compiler verworfen.

Kommentare mit Stil

Alle Kommentare und Bemerkungen sollten in Englisch verfasst werden, um Projektmitgliedern aus anderen Ländern das Lesen zu erleichtern. Für allgemeine Kommentare sollten wir die Zeichen // benutzen. Sie haben zwei Vorteile:

- Bei Editoren, die Kommentare nicht farbig hervorheben, oder bei einer einfachen Quellcodeausgabe auf der Kommandozeile lässt sich ersehen, dass eine Zeile, die mit // beginnt, ein Kommentar ist. Den Überblick über einen Quelltext zu behalten, der für mehrere Seiten mit den Kommentarzeichen /* und */ unterbrochen wird, ist schwierig. Zeilenkommentare machen deutlich, wo Kommentare beginnen und wo sie enden.

- Der Einsatz der Zeilenkommentare eignet sich besser dazu, während der Entwicklungs- und Debug-Phase Codeblöcke auszukommentieren. Benutzen wir zur Programmdokumentation die Blockkommentare, so sind wir eingeschränkt, denn Kommentare dieser Form können wir nicht verschachteln. Zeilenkommentare können einfacher geschachtelt werden.

Die Tastenkombination Strg + 7 – oder Strg + /, was das Kommentarzeichen »/« noch deutlicher macht – kommentiert eine Zeile aus. Eclipse setzt dann vor die Zeile die Kommentarzeichen //. Sind mehrere Zeilen selektiert, kommentiert die Tastenkombination alle markierten Zeilen mit Zeilenkommentaren aus. In einer kommentierten Zeile nimmt ein erneutes Strg + 7 die Kommentare einer Zeile wieder zurück.

Strg + ⇧ + C in kommentiert eine Zeile bzw. einen Block in NetBeans ein und aus.

2.2 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.

2.2.1 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`

Hinweis

Diese Befehlsform ist für Programmiersprachen gar nicht selbstverständlich, da es Sprachen gibt, die zu einer Problembeschreibung selbstständig eine Lösung finden. Ein Vertreter dieser Art von Sprachen ist *Prolog*.

Hinweis (Forts.)

Die Schwierigkeit hierbei besteht darin, die Aufgabe so präzise zu beschreiben, dass das System eine Lösung finden kann. Auch die Datenbanksprache SQL ist keine imperative Programmiersprache, denn wie das Datenbankmanagement-System zu unserer Abfrage die Ergebnisse ermittelt, müssen und können wir weder vorgeben noch sehen.

2.2.2 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
  }
}
```

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

Eclipse zeigt Schlüsselwörter, Literale und Kommentare farbig an. Diese Farbgebung lässt sich unter WINDOW • PREFERENCES ändern.

Unter TOOLS • OPTIONS und dann im Bereich FONTS & COLOR lassen sich bei NetBeans der Zeichensatz und die Farbbelegung ändern.

Java ist eine objektorientierte Programmiersprache, die Programmlogik außerhalb von Klassen nicht erlaubt. Aus diesem Grund deklariert die Datei *Application.java* mit dem Schlüsselwort `class` eine Klasse `Application`, um später eine Methode mit der Programmlogik anzugeben. Der Klassenname darf grundsätzlich beliebig sein, doch besteht die Einschränkung, dass in einer mit `public` deklarierten Klasse der Klassenname so lauten muss wie der Dateiname. Alle Schlüsselwörter in Java beginnen mit Kleinbuchstaben, und Klassennamen beginnen üblicherweise mit Großbuchstaben.

In den geschweiften Klammern der Klasse folgen Deklarationen von Methoden, also Unterprogrammen, die eine Klasse anbietet. Eine Methode ist eine Sammlung von Anweisungen unter einem Namen.

2.2.3 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.[7] 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.

eclipse

Dass Fehler unterkringelt werden, hat sich als Visualisierung durchgesetzt. Eclipse gibt im Falle eines Fehlers sehr viele Hinweise. Ein Fehler im Quellcode wird von Eclipse mit einer gekringelten roten Linie angezeigt. Als weiterer Indikator wird (unter Umständen erst beim Speichern) ein kleines rundes Kreuz an der Fehlerzeile angezeigt. Gleichzeitig findet sich im Schieberegler ein kleiner roter Block. Im Package Explorer findet sich ebenfalls ein Hinweis auf Fehler. Zum nächsten Fehler bringt die Tastenkombination Strg + . (Punkt) zurück.

7 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.

Strg + . führt bei NetBeans nur in der Fehleransicht zum Fehler selbst, nicht aber aus dem Java-Editor heraus. Die Tastenkombination kann im Editor einfach über Tools • Option gesetzt werden. Dann wählen wir Keymap, geben unter Search den Suchbegriff »Error« ein und selektieren dann in der Action-Spalte Next Error in Editor. In der zweiten Spalte Shortcut setzen wir den Fokus und drücken Strg + . . Dann beenden wir den Dialog mit OK.

2.2.4 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[8] 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
  }
}
```

8 »Implementieren« stammt vom lateinischen Wort »implere« ab, was für »erfüllen« und »ergänzen« steht.

Hinweis

Der Begriff *Methode* ist die korrekte Bezeichnung für ein Unterprogramm in Java – die *Java Language Specification* (JLS) verwendet den Begriff *Funktion* nicht.

2.2.5 Atomare Anweisungen und Anweisungssequenzen

Methodenaufrufe wie `System.out.println()`, die *leere Anweisung*, die nur aus einem Semikolon besteht, oder auch Variablendeklarationen (die später vorgestellt werden) nennen sich *atomare* (auch *elementare) Anweisungen*. Diese unteilbaren Anweisungen werden zu *Anweisungssequenzen* zusammengesetzt, die Programme bilden.

zB Beispiel

Eine Anweisungssequenz:

```
System.out.println( "Wer morgens total zerknittert aufsteht, " );
System.out.println( "hat am Tag die besten Entfaltungsmöglichkeiten." );
;
System.out.println();
;
```

Leere Anweisungen (also die Zeilen mit dem Semikolon) gibt es im Allgemeinen nur bei Endloswiederholungen.

Die Laufzeitumgebung von Java führt jede einzelne Anweisung der Sequenz in der angegebenen Reihenfolge hintereinander aus. Anweisungen und Anweisungssequenzen dürfen nicht irgendwo stehen, sondern nur an bestimmen Stellen, etwa innerhalb eines Methodenkörpers.

2.2.6 Mehr zu print(), println() und printf() für Bildschirmausgaben

Die meisten Methoden verraten durch ihren Namen, was sie leisten, und für eigene Programme ist es sinnvoll, aussagekräftige Namen zu verwenden. Wenn die Java-Entwickler die Ausgabemethode statt `println()` einfach `glubschi()` genannt hätten, bliebe uns der Sinn der Methode verborgen. `println()` zeigt jedoch durch den Wortstamm »print« an, dass etwas geschrieben wird. Die Endung `ln` (kurz für *line*) bedeutet, dass noch ein Zeilenvorschubzeichen ausgegeben wird. Umgangssprachlich heißt das: Eine neue Aus-

gabe beginnt in der nächsten Zeile. Neben `println()` existiert die Bibliotheksmethode `print()`, die keinen Zeilenvorschub anhängt.

Die `printXXX()`-Methoden[9] können in Klammern unterschiedliche Argumente bekommen. Ein Argument ist ein Wert, den wir der Methode beim Aufruf mitgeben. Auch wenn wir einer Methode keine Argumente übergeben, muss beim Aufruf hinter dem Methodennamen ein Klammernpaar folgen. Dies ist konsequent, da wir so wissen, dass es sich um einen Methodenaufruf handelt und um nichts anderes. Andernfalls führt es zu Verwechslungen mit Variablen.

Überladene Methoden

Java erlaubt Methoden, die gleich heißen, denen aber unterschiedliche Dinge übergeben werden können; diese Methoden nennen wir *überladen*. Die `printXXX()`-Methoden sind zum Beispiel überladen und akzeptieren neben dem Argumenttyp `String` auch Typen wie einzelne Zeichen, Wahrheitswerte oder Zahlen – oder auch gar nichts:

Listing 2.3: OverloadedPrintln.java

```
public class OverloadedPrintln
{
  public static void main( String[] args )
  {
    System.out.println( "Verhaften Sie die üblichen Verdächtigen!" );
    System.out.println( true );
    System.out.println( -273 );
    System.out.println();               // Gibt eine Leerzeile aus
    System.out.println( 1.6180339887498948 );
  }
}
```

Die Ausgabe ist:

```
Verhaften Sie die üblichen Verdächtigen!
true
-273

1.618033988749895
```

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

In der letzten Zeile ist gut zu sehen, dass es Probleme mit der Genauigkeit gibt – dieses Phänomen werden wir uns noch genauer anschauen.

Ist in Eclipse eine andere Ansicht aktiviert, etwa weil wir auf das Konsolenfenster geklickt haben, bringt die Taste F12 uns wieder in den Editor zurück.

Variable Argumentlisten

Java unterstützt seit der Version 5 variable Argumentlisten, was bedeutet, dass es möglich ist, bestimmten Methoden beliebig viele Argumente (oder auch kein Argument) zu übergeben. Die Methode `printf()` erlaubt zum Beispiel variable Argumentlisten, um gemäß einer Formatierungsanweisung – einem String, der immer als erstes Argument übergeben werden muss – die nachfolgenden Methodenargumente aufzubereiten und auszugeben:

Listing 2.4: VarArgs.java

```
public class VarArgs
{
  public static void main( String[] args )
  {
    System.out.printf( "Was sagst du?%n" );
    System.out.printf( "%d Kanäle und überall nur %s.%n", 220, "Katzen" );
  }
}
```

Die Ausgabe der Anweisung ist:

```
Was sagst du?
220 Kanäle und überall nur Katzen.
```

Die Formatierungsanweisung `%n` setzt einen Zeilenumbruch, `%d` ist ein Platzhalter für eine Dezimalzahl und `%s` ein Platzhalter für eine Zeichenkette oder etwas, das in einen String konvertiert werden soll. Weitere Platzhalter werden in Abschnitt 4.11, »Ausgaben formatieren«, vorgestellt.

2.2.7 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.

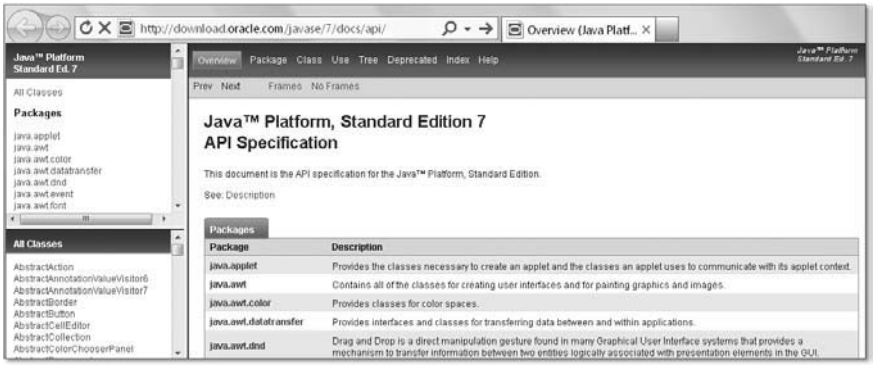

Abbildung 2.1: Online-Dokumentation bei Oracle

Aus Entwicklungsumgebungen ist die API-Dokumentation auch zugänglich, sodass eine Suche auf der Webseite nicht nötig ist.

Eclipse zeigt mithilfe der Tasten ⇧ + F2 in einem eingebetteten Browser-Fenster die API-Dokumentation an, wobei die JavaDoc von den Oracle-Seiten kommt. Mithilfe der F2-Taste bekommen wir ein kleines gelbes Vorschaufenster, das ebenfalls die API-Dokumentation zeigt.

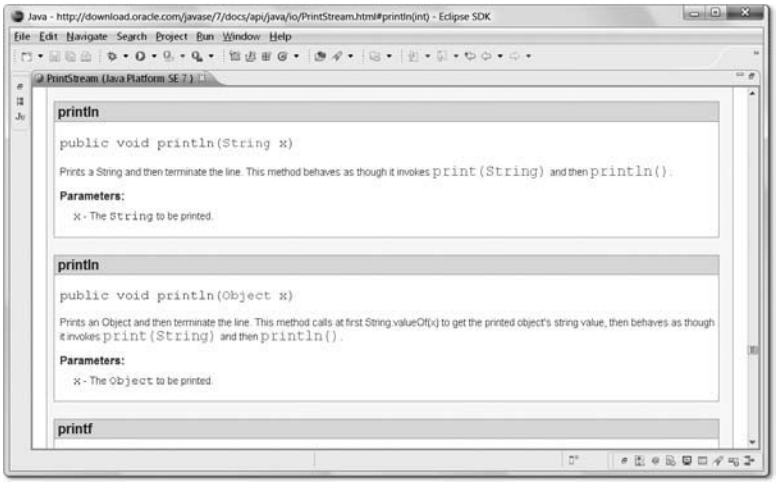

Abbildung 2.2: API-Dokumentation in Eclipse

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.

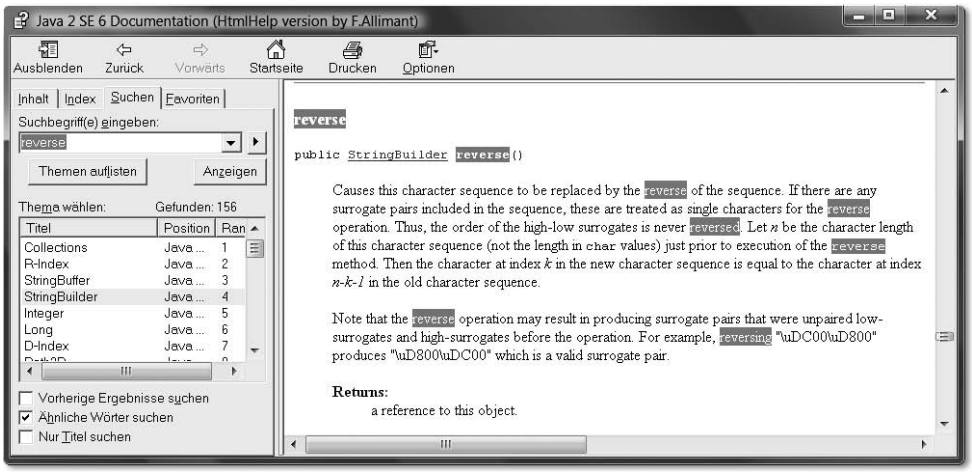

Abbildung 2.3: API-Dokumentation in der Windows-Hilfe

2.2.8 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).

2.2.9 Ausdrucksanweisung

In einem Programm reiht sich Anweisung an Anweisung. Auch bestimmte Ausdrücke und Methodenaufrufe lassen sich als Anweisungen einsetzen, wenn sie mit einem Semikolon abgeschlossen sind; wir sprechen dann von einer *Ausdrucksanweisung* (engl. *expression statement*). Jeder Methodenaufruf mit Semikolon bildet zum Beispiel eine Ausdrucksanweisung. Dabei ist es egal, ob die Methode selbst eine Rückgabe liefert oder nicht.

```
System.out.println();        // println() besitzt keine Rückgabe (void)
Math.random();               // random() liefert eine Fließkommazahl
```

Die Methode `Math.random()` liefert als Ergebnis einen Zufallswert zwischen 0 (inklusiv) und 1 (exklusiv). Da mit dem Ergebnis des Ausdrucks nichts gemacht wird, wird der Rückgabewert verworfen. Im Fall der Zufallsmethode ist das nicht sinnvoll, denn sie macht außer der Berechnung nichts anderes.

Neben Methodenaufrufen mit abschließendem Semikolon gibt es andere Formen von Ausdrucksanweisungen, wie etwa Zuweisungen. Doch allen ist das Semikolon gemeinsam.[10]

Hinweis

Nicht jeder Ausdruck kann eine Ausdrucksanweisung sein. `1+2` ist etwa ein Ausdruck, aber `1+2;` – also der Ausdruck mit Semikolon abgeschlossen – ist keine gültige Anweisung. In JavaScript ist so etwas erlaubt, in Java nicht.

10 Das Semikolon dient auch nicht wie in Pascal zur Trennung von Anweisungen, sondern schließt sie immer ab.

2.2.10 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.

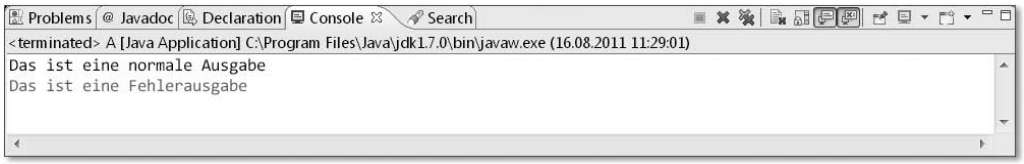

Abbildung 2.4: Eclipse stellt normale Ausgaben schwarz und Fehlerausgaben rot dar. Damit ist es leicht, zu erkennen, welche Ausgabe in welchen Kanal geschickt wurde.

2.2.11 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.

zB

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.

2.2.12 Gruppieren von Anweisungen mit Blöcken

Ein *Block* fasst eine Gruppe von Anweisungen zusammen, die hintereinander ausgeführt werden. Anders gesagt: Ein Block *ist eine Anweisung*, die in geschweiften Klammern { } eine Folge von Anweisungen zu einer neuen Anweisung zusammenfasst:

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

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

Codestyle

Die Zeilen, die in geschweiften Klammern stehen, werden in der Regel mit Leerraum eingerückt. Üblicherweise sind es zwei (wie in diesem Buch) oder vier Leerzeichen. Viele Autoren setzen die geschweiften Klammern in eine eigene Zeile. Diesem Stil folgt auch dieses Buch in der Regel, es sei denn, der Programmcode soll weniger »vertikal wachsen«.

Leerer Block

Ein Block {} ohne Anweisung nennt sich *leerer Block*. Er verhält sich wie eine leere Anweisung, also wie ein Semikolon. In einigen Fällen ist der leere Block mit dem Semikolon wirklich austauschbar, in anderen Fällen erzwingt die Java-Sprache einen Block, der, falls es keine Anweisungen gibt, leer ist, anstatt hier auch ein Semikolon zu erlauben.

Geschachtelte Blöcke

Blöcke können beliebig geschachtelt werden. So ergeben sich innere Blöcke und äußere Blöcke:

```
{             // Beginn äußerer Block
  {           // Beginn innerer Block
  }           // Ende innerer Block
}             // Ende äußerer Block
```

Mit leeren Blöcken ist Folgendes in der statischen Methode `main()` in Ordnung:

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

Blöcke spielen eine wichtige Rolle beim Zusammenfassen von Anweisungen, die in Abhängigkeit von Bedingungen einmal oder mehrmals ausgeführt werden. Im Abschnitt 2.5 und 2.6 kommen wir darauf noch einmal praktisch zurück.

2.3 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.[11]

Hinweis

In Java muss der Datentyp einer Variablen zur Übersetzungszeit bekannt sein. Das nennt sich dann *statisch typisiert*. Das Gegenteil ist eine *dynamische Typisierung*, wie sie etwa JavaScript verwendet. Hier kann sich der Typ einer Variablen zur Laufzeit ändern, je nachdem, was die Variable enthält.

11 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.

2.3.1 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.

Typ	Belegung (Wertebereich)
`boolean`	`true` oder `false`
`char`	16-Bit-Unicode-Zeichen (0x0000 ... 0xFFFF)

Tabelle 2.5: Java-Datentypen und ihre Wertebereiche

Typ	Belegung (Wertebereich)
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

Tabelle 2.5: Java-Datentypen und ihre Wertebereiche (Forts.)

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.[12]

Detailwissen

Genau genommen sieht die Sprachgrammatik von Java keine negativen Zahlenliterale vor. Bei einer Zahl wie –1.2 oder –1 ist das Minus der unäre Operator und gehört nicht zur Zahl. Im Bytecode selbst sind die negativen Zahlen natürlich wieder abgebildet.

12 Es gibt bei Fließkommazahlen noch »Sonderzahlen«, wie *plus* oder *minus Unendlich*, aber dazu später mehr.

Die folgende Tabelle zeigt eine etwas andere Darstellung:

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

Tabelle 2.6: Java-Datentypen und ihre Größen und Formate

Hinweis

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

Zwei wesentliche Punkte zeichnen die primitiven Datentypen aus:

- Alle Datentypen haben eine festgesetzte Länge, die sich unter keinen Umständen ändert. Der Nachteil, dass sich bei einigen Hochsprachen die Länge eines Datentyps ändern kann, besteht in Java nicht. In den Sprachen C(++) bleibt dies immer unsicher, und die Umstellung auf 64-Bit-Maschinen bringt viele Probleme mit sich. Der Datentyp `char` ist 16 Bit lang.
- Die numerischen Datentypen `byte`, `short`, `int` und `long` sind vorzeichenbehaftet, Fließkommazahlen sowieso. Dies ist leider nicht immer praktisch, aber wir müssen stets daran denken. Probleme gibt es, wenn wir einem Byte zum Beispiel den Wert 240 zuweisen wollen, denn 240 liegt außerhalb des Wertebereichs, der von –128 bis 127 reicht. Ein `char` ist im Prinzip ein vorzeichenloser Ganzzahltyp.

Wenn wir also die numerischen Datentypen (lassen wir hier `char` außen vor) nach ihrer Größe sortieren wollten, könnten wir zwei Linien für Ganzzahlen und Fließkommazahlen aufbauen:

```
byte < short < int < long
       float < double
```

Hinweis

Die Klassen `Byte`, `Integer`, `Long`, `Short`, `Character`, `Double` und `Float` deklarieren die Konstanten `MAX_VALUE` und `MIN_VALUE`, die den größten und kleinsten zulässigen Wert des jeweiligen Wertebereichs bzw. die Grenzen der Wertebereiche der jeweiligen Datentypen angeben.

```
System.out.println( Byte.MIN_VALUE );       // -128
System.out.println( Byte.MAX_VALUE );       // 127
System.out.println( Character.MIN_VALUE );  // '\u0000'
System.out.println( Character.MAX_VALUE );  // '\uFFFF'
System.out.println( Double.MIN_VALUE );     // 4.9E-324
System.out.println( Double.MAX_VALUE );     // 1.7976931348623157E308
```

2.3.2 Variablendeklarationen

Mit Variablen lassen sich Daten speichern, die vom Programm gelesen und geschrieben werden können. Um Variablen zu nutzen, müssen sie deklariert (definiert[13]) 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
{
```

13 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*.

```
  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?
  }
}
```

Der Typname ist entweder ein einfacher Typ (wie `int`) oder ein Referenztyp. Viel schwieriger ist eine Deklaration nicht – kryptische Angaben wie in C gibt es in Java nicht.[14] Ein Variablenname (der dann Bezeichner ist) kann alle Buchstaben und Ziffern des Unicode-Zeichensatzes beinhalten, mit der Ausnahme, dass am Anfang des Bezeichners keine Ziffer stehen darf. Auch darf der Bezeichnername mit keinem reservierten Schlüsselwort identisch sein.

Mehrere Variablen kompakt deklarieren

Im oberen Beispiel sind zwei Variaben vom gleichen Typ: `isPresident` und `isVegetarian`.

```
boolean isPresident;
boolean isVegetarian;
```

Immer dann, wenn der Variablentyp der gleiche ist, lässt sich die Deklaration verkürzen: Variablen werden mit Komma getrennt.

```
boolean isPresident, isVegetarian;
```

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:

14 Das ist natürlich eine Anspielung auf C, in dem Deklarationen wie `char (*(*a[2])())[2]` möglich sind. Gut, dass es mit *cdecl* ein Programm zum »Vorlesen« solcher Definitionen gibt.

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;
  }
}
```

Wir haben gesehen, dass bei der Deklaration mehrerer Variablen gleichen Typs ein Komma die Bezeichner trennt. Das überträgt sich auch auf die Initialisierung. Ein Beispiel:

```
boolean sendSms = true,
        bungaBungaParty = true;
String person1 = "Silvio", person2 = "Ruby the Heart Stealer";
double x, y, bodyHeight = 183;
```

Die Zeilen deklarieren mehrere Variablen auf einen Schlag. `x` und `y` am Schluss bleiben uninitialisiert.

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.

2.3.3 Konsoleneingaben

Bisher haben wir zwei Methoden kennengelernt: `prinln()` und `random()`. Die Methode `println()` »hängt« am `System.out` bzw. `System.err`-Objekt und `random()` »hängt« am `Math`-Objekt.

Der Gegenpol zu `println()` ist eine Konsoleneingabe. Hier gibt es unterschiedliche Varianten. Die einfachste ist mit der Klasse `java.util.Scanner`. In Kapitel 4, »Der Umgang mit Zeichenketten«, wird die Klasse noch viel genauer untersucht. Es reicht aber an dieser Stelle zu wissen, wie Strings, Ganzzahlen und Fließkommazahlen eingelesen werden.

Eingabe lesen vom Typ	Anweisung
`String`	`String s = new java.util.Scanner(System.in).nextLine();`
`int`	`int i = new java.util.Scanner(System.in).nextInt();`
`double`	`double d = new java.util.Scanner(System.in).nextDouble();`

Tabelle 2.7: Einlesen einer Zeichenkette, Ganz- und Fließkommazahl von der Konsole

Verbinden wir die drei Möglichkeiten zu einem Beispiel. Zunächst soll der Name eingelesen werden, dann das Alter und anschließend eine Fließkommazahl.

Listing 2.8: SmallConversation.java

```
public class SmallConversation
{
  public static void main( String[] args )
  {
    System.out.println( "Moin! Wie heißt denn du?" );
    String name = new java.util.Scanner( System.in ).nextLine();
    System.out.printf( "Hallo %s. Wie alt bist du?%n", name );
```

```
    int age = new java.util.Scanner( System.in ).nextInt();
    System.out.printf( "Aha, %s Jahre, das ist ja die Hälfte von %s.%n",
                       age, age * 2 );
    System.out.println( "Sag mal, was ist deine Lieblingsfließkommazahl?" );
    double value = new java.util.Scanner( System.in ).nextDouble();
    System.out.printf( "%s? Aha, meine ist %s.%n",
                       value, Math.random() * 100000 );
  }
}
```

Eine Konversation sieht somit etwa so aus:

```
Moin! Wie heißt denn du?
Christian
Hallo Christian. Wie alt bist du?
37
Aha, 37 Jahre, das ist ja die Hälfte von 74.
Sag mal, was ist deine Lieblingsfließkommazahl?
9,7
9.7? Aha, meine ist 60769.81705995359.
```

Die Eingabe der Fließkommazahl muss mit Komma erfolgen, wenn die JVM auf einem deutschsprachigen Betriebssystem läuft. Die Ausgabe über `printf()` kann ebenfalls lokalisierte Fließkommazahlen schreiben, dann muss jedoch statt dem Platzhalter `%s` die Kennung `%f` oder `%g` verwendet werden. Das wollen wir in einem zweiten Beispiel nutzen.

Zinsberechnung mit der Benutzereingabe

Die Zinsberechnung, die vorher feste Werte im Programm hatte, soll eine Benutzereingabe bekommen. Des Weiteren erwarteten wir die Dauer in Monaten statt Jahren.

Listing 2.9: MyInterestRates.java

```
public class MyInterestRates
{
  public static void main( String[] args )
  {
    System.out.println( "Kapital?" );
    double capital = new java.util.Scanner( System.in ).nextDouble();
```

```
    System.out.println( "Zinssatz?" );
    double interestRate = new java.util.Scanner( System.in ).nextDouble();

    System.out.println( "Anlagedauer in Monaten?" );
    int month = new java.util.Scanner( System.in ).nextInt();

    double interestRates = capital * interestRate * month / (12*100);
    System.out.printf( "Zinsen: %g%n", interestRates );
  }
}
```

Die vorher fest verdrahteten Werte sind nun alle dynamisch, und wir kommen mit den Eingaben zum gleichen Ergebnis wir vorher:

```
Kapital?
20000
Zinssatz?
3,6
Anlagedauer in Monaten?
24
Zinsen: 1440,00
```

2.3.4 Fließkommazahlen mit den Datentypen float und double

Für Fließkommazahlen (auch *Gleitkommazahlen* genannt) einfacher und erhöhter Genauigkeit bietet Java die Datentypen `float` und `double`. Die Datentypen sind im *IEEE 754-Standard* beschrieben und haben eine Länge von 4 Byte für `float` und 8 Byte für `double`. Fließkommaliterale können einen Vorkommateil und einen Nachkommateil besitzen, die durch einen Dezimalpunkt (kein Komma) getrennt sind. Ein Fließkommaliteral muss keine Vor- oder Nachkommastellen besitzen, sodass auch Folgendes gültig ist:

```
double d = 10.0 + 20. + .11;
```

Hinweis

Der Datentyp `float` ist mit 4 Byte, also 32 Bit, ein schlechter Scherz. Der Datentyp `double` geht mit 64 Bit ja gerade noch, wobei in Hardware eigentlich 80 Bit üblich sind.

Der Datentyp float *

Standardmäßig sind die Fließkomma-Literale vom Typ `double`. Ein nachgestelltes »f« (oder »F«) zeigt an, dass es sich um ein `float` handelt.

Beispiel zB

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

```
double pi = 3.1415, delta = .001;
float  ratio  = 4.33F;
```

Auch für den Datentyp `double` lässt sich ein »d« (oder »D«) nachstellen, was allerdings nicht nötig ist, wenn Literale für Kommazahlen im Quellcode stehen; Zahlen wie `3.1415` sind automatisch vom Typ `double`. Während jedoch bei `1 + 2 + 4.0` erst `1` und `2` als `int` addiert werden, dann in `double` und anschließend auf `4.0` addiert werden, würde `1D + 2 + 4.0` gleich mit der Fließkommazahl `1` beginnen. So ist auch `1D` gleich `1.` bzw. `1.0`.

Frage !

Was ist das Ergebnis der Ausgabe?

```
System.out.println( 20000000000F == 20000000000F+1 );
System.out.println( 20000000000D == 20000000000D+1 );
```

Tipp: Was sind die Wertebereiche von `float` und `double`?

Noch genauere Auflösung bei Fließkommazahlen *

Einen höher auflösenden beziehungsweise präziseren Datentyp für Fließkommazahlen als `double` gibt es nicht. Die Standardbibliothek bietet für diese Aufgabe in `java.math` die Klasse `BigDecimal` an, die in Kapitel 18, »Bits und Bytes und Mathematisches«, näher beschrieben ist. Das ist sinnvoll für Daten, die eine sehr gute Genauigkeit aufweisen sollen, wie zum Beispiel Währungen.[15]

15 Einige Programmiersprachen besitzen für Währungen eingebaute Datentypen, wie LotusScript mit `Currency`, das mit 8 Byte einen sehr großen und genauen Wertebereich abdeckt. Erstaunlicherweise gab es einmal in C# den Datentyp `currency` für ganzzahlige Währungen.

> **Sprachvergleich**
>
> In C# gibt es den Datentyp `decimal`, der mit 128 Bit (also 16 Byte) auch genügend Präzision bietet, um eine Zahl wie 0,00000000000000000000000001 auszudrücken.

2.3.5 Ganzzahlige Datentypen

Java stellt fünf ganzzahlige Datentypen zur Verfügung: `byte`, `short`, `char`, `int` und `long`. Die feste Länge von jeweils 1, 2, 2, 4 und 8 Byte ist eine wesentliche Eigenschaft von Java. Ganzzahlige Typen sind in Java immer vorzeichenbehaftet (mit der Ausnahme von `char`); einen Modifizierer `unsigned` wie in C(++) gibt es nicht.[16] Negative Zahlen werden durch Voranstellen eines Minuszeichens gebildet. Ein Pluszeichen für positive Zeichen ist möglich. `int` und `long` sind die bevorzugten Typen. `byte` kommt selten vor und `short` nur in wirklich sehr seltenen Fällen, etwa bei Feldern mit Bilddaten.

Ganzzahlen sind standardmäßig vom Typ int

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

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

Dennoch übersetzt der Compiler die Zeile nicht, da er ein Ganzzahlliteral ohne explizite Größenangabe als 32 Bit langes `int` annimmt. Die obige Zeile führt daher zu einem Compilerfehler, da unsere Zahl nicht im Wertebereich von –2.147.483.648 ... +2.147.483.647 liegt, sondern weit außerhalb: 2147483647 < 123456789012345. Java reserviert also *nicht* so viele Bits wie benötigt und wählt nicht automatisch den passenden Wertebereich.

Wer wird mehrfacher Milliardär? Der Datentyp long

Der Compiler betrachtet jede Ganzzahl automatisch als `int`. Sollte der Wertebereich von etwa plus/minus zwei Milliarden nicht reichen, greifen Entwickler zum nächsthöheren Datentyp. Dass eine Zahl `long` ist, muss ausdrücklich angegeben werden. Dazu wird an das Ende von Ganzzahlliteralen vom Typ `long` ein »l« oder »L« gesetzt. Um die Zahl `123456789012345` gültig ausgeben zu lassen, ist Folgendes zu schreiben:

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

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

Tipp

Das kleine »l« hat sehr viel Ähnlichkeit mit der Ziffer Eins. Daher sollte bei Längenangaben immer ein großes »L« eingefügt werden.

Frage

Was gibt die folgende Anweisung aus?

```
System.out.println( 123456789 + 5432l );
```

Der Datentyp byte

Ein `byte` ist ein Datentyp mit einem Wertebereich von –128 bis +127. Eine Initialisierung wie

```
byte b = 200;       // ☠
```

ist also nicht erlaubt, da 200 > 127 ist. Somit fallen alle Zahlen von 128 bis 255 (hexadezimal 80_{16} – FF_{16}) raus. In der Datenverarbeitung ist das Java-`byte`, weil es ein Vorzeichen trägt, nur mittelprächtig brauchbar, da insbesondere in der Dateiverarbeitung Wertebereiche von 0 bis 255 gewünscht sind.

Java erlaubt zwar keine vorzeichenlosen Ganzzahlen, aber mit zwei Schreibweisen lassen sich doch Zahlen wie 200 in einem `byte` speichern.

```
byte b = (byte) 200;
```

Der Java-Compiler nimmt dazu einfach die Bitbelegung von 200 und interpretiert das oberste dann gesetzte Bit als Vorzeichenbit. Bei der Ausgabe fällt das auf:

```
byte b = (byte) 200;
System.out.println( b );      // -56
```

Der Datentyp short *

Der Datentyp `short` ist selten anzutreffen. Mit seinen 2 Byte kann er einen Wertebereich von –32.768 bis +32.767 darstellen. Das Vorzeichen »kostet« wie bei den anderen Ganzzahlen 1 Bit, sodass nicht 16 Bit, sondern nur 15 Bit für Zahlen zu Verfügung stehen. Allerdings gilt wie beim `byte`, dass auch ein `short` ohne Vorzeichen auf zwei Arten initialisiert werden kann:

```
short s = (short) 33000;
System.out.println( s );      // -32536
```

2.3.6 Wahrheitswerte

Der Datentyp `boolean` beschreibt einen Wahrheitswert, der entweder `true` oder `false` ist. Die Zeichenketten `true` und `false` sind reservierte Wörter und bilden neben konstanten Strings und primitiven Datentypen Literale. Kein anderer Wert ist für Wahrheitswerte möglich, insbesondere werden numerische Werte nicht als Wahrheitswerte interpretiert.

Der boolesche Typ wird beispielsweise bei Bedingungen, Verzweigungen oder Schleifen benötigt. In der Regel ergibt sich ein Wahrheitswert aus Vergleichen.

2.3.7 Unterstriche in Zahlen *

Um eine Anzahl von Millisekunden in Tage zu konvertieren, muss einfach eine Division vorgenommen werden. Um Millisekunden in Sekunden umzurechnen, brauchen wir eine Division durch 1000, von Sekunden auf Minuten eine Division durch 60, von Minuten auf Stunden eine Division durch 60, und die Stunden auf Tage bringt die letzte Division durch 24. Schreiben wir das auf:

```
long millis = 10 * 24*60*60*1000L;
long days  = millis / 86400000L;
System.out.println( days );    // 10
```

Eine Sache fällt bei der Zahl 86400000 auf: Besonders gut lesbar ist sie nicht. Die eine Lösung ist, es erst gar nicht zu so einer Zahl kommen zu lassen und sie wie in der ersten Zeile durch eine Reihe von Multiplikationen aufzubauen – mehr Laufzeit kostet das nicht, da dieser konstante Ausdruck zur Übersetzungszeit feststeht.

Die zweite Variante ist eine neue Schreibweise, die Java 7 einführt: *Unterstriche in Zahlen*. Anstatt ein numerisches Literal als 86400000 zu schreiben, ist in Java 7 auch Folgendes erlaubt:

```
long millis = 10 * 86_400_000L;
long days  = millis / 86_400_000L;
System.out.println( days );    // 10
```

Die Unterstriche machen die 1000er-Blöcke gut sichtbar. Hilfreich ist die Schreibweise auch bei Literalen in Binär- und Hexdarstellung, da die Unterstriche hier ebenfalls Blöcke absetzen können.[17]

Beispiel zB

Unterstriche verdeutlichen Blöcke bei Binär- und Hexadezimalzahlen.

```
int  i = 0b01101001_01001101_11100101_01011110;
long l = 0x7fff_ffff_ffff_ffffL;
```

Der Unterstrich darf in jedem Literal stehen, zwei aufeinanderfolgende Unterstriche sind aber nicht erlaubt.

2.3.8 Alphanumerische Zeichen

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

Beispiel zB

Korrekte Hochkommata für Zeichen und Zeichenketten:

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

Da der Compiler ein `char` automatisch in ein `int` konvertieren kann, ist auch `int c = 'a';` gültig.

17 Bei Umrechnungen zwischen Stunden, Minuten und so weiter hilft auch die Klasse `TimeUnit` mit einigen statischen `toXXX()`-Methoden.

2.3.9 Gute Namen, schlechte Namen

Für die optimale Lesbarkeit und Verständlichkeit eines Programmcodes sollten Entwickler beim Schreiben einige Punkte berücksichtigen:

- **Ein konsistentes Namensschema ist wichtig.** Heißt ein Zähler `no`, `nr`, `cnr` oder `counter`? Auch sollten wir korrekt schreiben und auf Rechtschreibfehler achten, denn leicht wird aus `necessaryConnection` dann `nesesarryConnection`. Variablen ähnlicher Schreibweise, etwa `counter` und `counters`, sind zu vermeiden.
- **Abstrakte Bezeichner sind ebenfalls zu vermeiden.** Die Deklaration `int TEN = 10;` ist absurd. Eine unsinnige Idee ist auch die folgende: `boolean FALSE = true, TRUE = false;`. Im Programmcode würde dann mit `FALSE` und `TRUE` gearbeitet. Einer der obersten Plätze bei einem Wettbewerb für die verpfuschtesten Java-Programme wäre uns gewiss.
- **Unicode-Sequenzen können zwar in Bezeichnern aufgenommen werden, doch sollten sie vermieden werden.** In `double übelkübel, \u00FCbelk\u00FCbel;` sind beide Bezeichnernamen gleich, und der Compiler meldet einen Fehler.
- **0 und O und 1 und l sind leicht zu verwechseln.** Die Kombination »rn« ist schwer zu lesen und je nach Zeichensatz leicht mit »m« zu verwechseln.[18] Gültig – aber böse – ist auch: `int ínt, ìnt, înt; boolean bôõleañ;`

> **! Bemerkung**
>
> In China gibt es 90 Millionen Familien mit dem Nachnamen Li. Das wäre so, als ob wir jede Variable `temp1`, `temp2` ... nennen würden.

eclipse

Ist ein Bezeichnername unglücklich gewählt (`pneumonoultramicroscopicsilicovolcanoconiosis` ist schon etwas lang), so lässt er sich problemlos konsistent umbenennen. Dazu wählen wir im Menü Refactor • Rename – oder auch kurz Alt + ⇧ + R; der Cursor muss auf dem Bezeichner stehen. Eine optionale Vorschau (engl. *preview*) zeigt an, welche Änderungen die Umbenennung nach sich ziehen wird. Neben Rename gibt es auch noch eine andere Möglichkeit. Dazu lässt sich auf der Variablen mit Strg + 1 ein Popup-Fenster mit Local Rename öffnen. Der Bezeichner wird selektiert und lässt sich ändern. Gleichzeitig ändern sich alle Bezüge auf die Variable mit.

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

2.3.10 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, anderfalls gibt der Compiler eine Fehlermeldung aus.[19]

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.

Eclipse zeigt einen Hinweis und einen Verbesserungsvorschlag an, wenn eine lokale Variable nicht initialisiert ist.

19 Anders ist das bei Objektvariablen (und statischen Variablen sowie Feldern). Sie sind standardmäßig mit `null` (Referenzen), `0` (bei Zahlen) oder `false` belegt.

Kapitel 11
Die Klassenbibliothek

»Was wir brauchen, sind ein paar verrückte Leute;
seht euch an, wohin uns die Normalen gebracht haben.«
– George Bernard Shaw (1856–1950)

11.1 Die Java-Klassenphilosophie

Eine Programmiersprache besteht nicht nur aus einer Grammatik, sondern, wie im Fall von Java, auch aus einer Programmierbibliothek. Eine plattformunabhängige Sprache – so wie sich viele C oder C++ vorstellen – ist nicht wirklich plattformunabhängig, wenn auf jedem Rechner andere Funktionen und Programmiermodelle eingesetzt werden. Genau dies ist der Schwachpunkt von C(++). Die Algorithmen, die kaum vom Betriebssystem abhängig sind, lassen sich überall gleich anwenden, doch spätestens bei grafischen Oberflächen ist Schluss. Dieses Problem ergibt sich in Java seltener, weil sich die Entwickler große Mühe gaben, alle wichtigen Methoden in wohlgeformten Klassen und Paketen unterzubringen. Diese decken insbesondere die zentralen Bereiche Datenstrukturen, Ein- und Ausgabe, Grafik- und Netzwerkprogrammierung ab.

11.1.1 Übersicht über die Pakete der Standardbibliothek

Die Java 7-Klassenbibliothek bietet genau 208 Pakete.[1] Die wichtigsten davon fasst die folgende Tabelle zusammen:

1 Unsere Kollegen aus der Microsoft-Welt müssen eine dickere Pille schlucken, denn .NET 4 umfasst 408 Pakete (*Assemblies* genannt). Dafür enthält .NET aber auch Dinge, die in der Java-Welt der Java EE zuzuordnen sind. Aber auch dann liegt .NET immer noch vorne, denn Java EE 6 deklariert gerade einmal 117 Pakete.

Paket	Beschreibung
`java.awt`	Das Paket AWT (*Abstract Windowing Toolkit*) bietet Klassen zur Grafikausgabe und zur Nutzung von grafischen Bedienoberflächen.
`java.awt.event`	Schnittstellen für die verschiedenen Ereignisse unter grafischen Oberflächen
`java.io`	Möglichkeiten zur Ein- und Ausgabe. Dateien werden als Objekte repräsentiert. Datenströme erlauben den sequenziellen Zugriff auf die Dateiinhalte.
`java.lang`	Ein Paket, das automatisch eingebunden ist und unverzichtbare Klassen wie String-, Thread- oder Wrapper-Klassen enthält
`java.net`	Kommunikation über Netzwerke. Bietet Klassen zum Aufbau von Client- und Serversystemen, die sich über TCP beziehungsweise IP mit dem Internet verbinden lassen.
`java.text`	Unterstützung für internationalisierte Programme. Bietet Klassen zur Behandlung von Text und zur Formatierung von Datumswerten und Zahlen.
`java.util`	Bietet Typen für Datenstrukturen, Raum und Zeit sowie für Teile der Internationalisierung sowie für Zufallszahlen.
`javax.swing`	Swing-Komponenten für grafische Oberflächen. Das Paket besitzt diverse Unterpakete.

Tabelle 11.1: Wichtige Pakete in Java 7

Eine vollständige Übersicht aller Pakete gibt Anhang A, »Die Klassenbibliothek«. Als Entwickler ist es unumgänglich für die Details die JavaDoc unter *http://download.oracle.com/javase/7/docs/api/* zu studieren.

Offizielle Schnittstelle (java und javax-Pakete)

Das, was die JavaDoc dokumentiert, bildet den erlaubten Zugang zum JDK. Die Typen sind für die Ewigkeit ausgelegt, sodass Entwickler darauf zählen können, auch noch in 100 Jahren ihre Java-Programme ausführen zu können. Doch wer definiert die API? Im Kern sind es vier Quellen:

- Oracle-Entwickler setzen neue Pakete und Typen in die API.
- Der *Java Community Process* (JCP) beschließt eine neue API. Dann ist es nicht nur Oracle allein, sondern eine Gruppe, die eine neue API erarbeitet und die Schnittstellen definiert.

- Die *Object Management Group* (OMG) definiert eine API für CORBA.
- Das *World Wide Web Consortium* (W3C) gibt eine API etwa für XML-DOM vor.

Die Merkhilfe ist, dass alles, was mit `java` oder `javax` beginnt, eine erlaubte API darstellt, und alles andere zu nicht portablen Java-Programmen führen kann. Es gibt weiterhin Klassen, die unterstützt werden, aber nicht Teil der offiziellen API sind. Dazu zählen etwa diverse Swing-Klassen für das Aussehen der Oberfläche.

Hinweis

Die Laufzeitumgebung von Oracle liefert noch über 3.000 Klassendateien in den Paketen `sun` und `sunw` aus. Diese internen Klassen sind nicht offiziell dokumentiert,[2] aber zum Teil sehr leistungsfähig und erlauben selbst direkten Speicherzugriff oder können Objekte ohne Standard-Konstruktor erzeugen:

Listing 11.1: com/tutego/insel/sun/UnsafeInstance.java, Ausschnitt

```
Field field = Unsafe.class.getDeclaredField( "theUnsafe" );
field.setAccessible( true );
sun.misc.Unsafe unsafe = (sun.misc.Unsafe) field.get( null );
File f = (File) unsafe.allocateInstance( File.class );
System.out.println( f.getPath() );                          // null
```

`File` hat keinen Standard-Konstruktor, nicht einmal einen privaten. Diese Art der Objekterzeugung kann bei der Deserialisierung hilfreich sein.

Standard Extension API (javax-Pakete)

Einige der Java-Pakete beginnen mit `javax`. Dies sind ursprünglich Erweiterungspakete (E**x**tensions), die die Kern-Klassen ergänzen sollten. Im Laufe der Zeit sind jedoch viele der früher zusätzlich einzubindenden Pakete in die Standard-Distribution gewandert, sodass heute ein recht großer Anteil mit `javax` beginnt, aber keine Erweiterungen mehr darstellt, die zusätzlich installiert werden müssen. Sun wollte damals die Pakete nicht umbenennen, um so eine Migration nicht zu erschweren. Fällt heute im Quellcode ein Paketname mit `javax` auf, ist es daher nicht mehr so einfach zu entscheiden, ob eine externe Quelle mit eingebunden werden muss beziehungsweise ab welcher Java-Version das Paket Teil der Distribution ist. Echte externe Pakete sind unter anderem:

2 Das Buch »Java Secrets« von Elliotte Rusty Harold, *http://ibiblio.org/java/books/secrets/*, IDG Books, ISBN 0764580078, geht einigen Klassen nach, ist aber schon älter.

- *Enterprise/Server API* mit den Enterprise *JavaBeans, Servlets* und *JavaServer Faces*
- *Java Persistence API* (JPA) zum dauerhaften Abbilden von Objekten auf (in der Regel) relationale Datenbanken
- *Java Communications API* für serielle und parallele Schnittstellen
- *Java Telephony API*
- Sprachein-/-ausgabe mit der *Java Speech API*
- *JavaSpaces* für gemeinsamen Speicher unterschiedlicher Laufzeitumgebungen
- *JXTA* zum Aufbauen von P2P-Netzwerken

11.2 Sprachen der Länder

Programme der ersten Generation konnten nur mit fest verdrahteten Landessprachen und landesüblichen Bezeichnungen umgehen. Daraus ergaben sich natürlich vielfältige Probleme. Mehrsprachige Programme mussten aufwendig entwickelt werden, damit sie unter mehreren Sprachen lokalisierte Ausgaben lieferten. (Es ergaben sich bereits Probleme durch unterschiedliche Zeichenkodierungen. Dies umging aber der Unicode-Standard.) Es blieb das Problem, dass sprachabhängige Zeichenketten, wie alle anderen Zeichenketten auch, überall im Programmtext verteilt sind und eine nachträgliche Sprachanpassung nur aufwendig zu erreichen ist. Java bietet hier eine Lösung an: zum einen durch die Definition einer Sprache und damit durch automatische Formatierungen, und zum anderen durch die Möglichkeit, sprachenabhängige Teile in Ressourcen-Dateien auszulagern.

11.2.1 Sprachen und Regionen über Locale-Objekte

In Java repräsentieren `Locale`-Objekte geografische, politische oder kulturelle Regionen. Die Sprache und die Region müssen getrennt werden, denn nicht immer gibt eine Region oder ein Land die Sprache eindeutig vor. Für Kanada in der Umgebung von Quebec ist die französische Ausgabe relevant, und die unterscheidet sich von der englischen. Jede dieser sprachspezifischen Eigenschaften ist in einem speziellen Objekt gekapselt.

zB **Beispiel**

Sprach-Objekte werden immer mit dem Namen der Sprache und optional mit dem Namen des Landes beziehungsweise einer Region erzeugt. Im Konstruktor der Klasse `Locale` werden dann Länderabkürzungen angegeben, etwa für ein Sprach-Objekt für Großbritannien oder Frankreich:

Beispiel (Forts.) (Forts.) zB

```
Locale greatBritain = new Locale( "en", "GB" );
Locale french       = new Locale( "fr" );
```

Im zweiten Beispiel ist uns das Land egal. Wir haben einfach nur die Sprache Französisch ausgewählt, egal in welchem Teil der Welt.

Die Sprachen sind durch Zwei-Buchstaben-Kürzel aus dem ISO-639-Code[3] (ISO Language Code) identifiziert, und die Ländernamen sind Zwei-Buchstaben-Kürzel, die in ISO 3166[4] (ISO Country Code) beschrieben sind.

```
final class java.util.Locale
implements Cloneable, Serializable
```


- `Locale(String language)`
  Erzeugt ein neues `Locale`-Objekt für die Sprache (`language`), die nach dem ISO-693-Standard gegeben ist.
- `Locale(String language, String country)`
  Erzeugt ein `Locale`-Objekt für eine Sprache (`language`) nach ISO 693 und ein Land (`country`) nach dem ISO-3166-Standard.
- `public Locale(String language, String country, String variant)`
  Erzeugt ein `Locale`-Objekt für eine Sprache, ein Land und eine Variante. `variant` ist eine herstellerabhängige Angabe wie »WIN« oder »MAC«.

Die statische Methode `Locale.getDefault()` liefert die aktuell eingestellte Sprache. Für die laufende JVM kann `Locale.setLocale(Locale)` diese ändern.

Konstanten für einige Länder und Sprachen

Die `Locale`-Klasse besitzt Konstanten für häufig auftretende Länder und Sprachen. Statt für Großbritannien explizit `new Locale("en", "GB")` zu schreiben, bietet die Klasse mit `Locale.UK` eine Abkürzung. Unter den Konstanten für Länder und Sprachen sind: `CANADA`, `CANADA_FRENCH`, `CHINA` ist gleich `CHINESE` (und auch `PRC` bzw. `SIMPLIFIED_CHINESE`), `ENGLISH`, `FRANCE`, `FRENCH`, `GERMAN`, `GERMANY`, `ITALIAN`, `ITALY`, `JAPAN`, `JAPANESE`, `KOREA`, `KOREAN`, `TAIWAN` (ist gleich `TRADITIONAL_CHINESE`), `UK` und `US`.

3 *http://www.loc.gov/standards/iso639-2/php/code_list.php*

4 *http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/index.html*

Methoden von Locale

`Locale`-Objekte bieten eine Reihe von Methoden an, um etwa den ISO-639-Code des Landes preiszugeben.

zB **Beispiel**

Gib für Deutschland zugängliche Informationen aus. Das Objekt `out` aus `System` und `GERMANY` aus `Locale` sind statisch importiert:

Listing 11.2: com/tutego/insel/locale/GermanyLocal.java, main()

```
out.println( GERMANY.getCountry() );         // DE
out.println( GERMANY.getLanguage() );        // de
out.println( GERMANY.getVariant() );         //
out.println( GERMANY.getDisplayCountry() );  // Deutschland
out.println( GERMANY.getDisplayLanguage() ); // Deutsch
out.println( GERMANY.getDisplayName()  );    // Deutsch (Deutschland)
out.println( GERMANY.getDisplayVariant() );  //
out.println( GERMANY.getISO3Country() );     // DEU
out.println( GERMANY.getISO3Language() );    // deu
```

```
final class java.util.Locale
implements Cloneable, Serializable
```

- `String getCountry()`
  Liefert das Länderkürzel nach dem ISO-3166-zwei-Buchstaben-Code.
- `String getLanguage()`
  Liefert das Kürzel der Sprache im ISO-639-Code.
- `String getVariant()`
  Liefert das Kürzel der Variante.
- `final String getDisplayCountry()`
  Liefert ein Kürzel des Landes für Bildschirmausgaben.
- `final String getDisplayLanguage()`
  Liefert ein Kürzel der Sprache für Bildschirmausgaben.
- `final String getDisplayName()`
  Liefert den Namen der Einstellungen.
- `final String getDisplayVariant()`
  Liefert den Namen der Variante.

- `String getISO3Country()`
  Liefert die ISO-Abkürzung des Landes dieser Einstellungen und löst eine `MissingResourceException` aus, wenn die ISO-Abkürzung nicht verfügbar ist.
- `String getISO3Language()`
  Liefert die ISO-Abkürzung der Sprache dieser Einstellungen und löst eine `MissingResourceException` aus, wenn die ISO-Abkürzung nicht verfügbar ist.
- `static Locale[] getAvailableLocales()`
  Liefert eine Aufzählung aller installierten `Locale`-Objekte. Das Feld enthält mindestens `Locale.US` und unter Java 7 fast 160 Einträge.

Abbildung 11.1: UML-Diagramm der Locale-Klasse

11.3 Die Klasse Date

Die ältere Klasse `java.util.Date` ist durch die Aufgabenverteilung auf die Klassen `DateFormat` und `Calendar` sehr schlank. Ein Exemplar der Klasse `Date` verwaltet ein besonderes Datum oder eine bestimmte Zeit; die Zeitgenauigkeit beträgt eine Millisekunde. `Date`-Objekte sind mutable, also veränderbar. Sie lassen sich daher nur mit Vorsicht an Methoden übergeben oder zurückgeben.

Im SQL-Paket gibt es eine Unterklasse von `java.util.Date`, die Klasse `java.sql.Date`. Bis auf eine statische Methode `java.sql.Date.valueOf(String)`, die Zeichenfolgen mit dem Aufbau »yyyy-mm-dd« erkennt, gibt es keine Unterschiede.

11.3.1 Objekte erzeugen und Methoden nutzen

Viele Methoden von `Date` sind veraltet, und zwei Konstruktoren der Klasse bleiben uns:

```
class java.util.Date
implements Serializable, Cloneable, Comparable<Date>
```

- `Date()`
  Erzeugt ein Datum-Objekt und initialisiert es mit der Zeit, die bei der Erzeugung gelesen wurde. Die gegenwärtige Zeit erfragt dieser Konstruktor mit `System.currentTimeMillis()`.
- `Date(long date)`
  Erzeugt ein Datum-Objekt und initialisiert es mit der übergebenen Anzahl von Millisekunden seit dem 1. Januar 1970, 00:00:00 GMT.

zB **Beispiel**

Mit der `toString()`-Methode können wir ein minimales Zeitanzeige-Programm schreiben. Wir rufen den Standard-Konstruktor auf und geben dann die Zeit aus. Die `println()`-Methode ruft wie üblich automatisch `toString()` auf:

Listing 11.3: com/tutego/insel/date/MiniClock.java

```
package com.tutego.insel.date;

class MiniClock
{
```

Beispiel (Forts.) zB

```
  public static void main( String[] args )
  {
    System.out.println( new java.util.Date() ); // Fri Jul 07 09:05:16 CEST 2006
  }
}
```

Die anderen Methoden erlauben Zeitvergleiche und operieren auf den Millisekunden.

```
class java.util.Date
implements Serializable, Cloneable, Comparable<Date>
```

- `long getTime()`
  Liefert die Anzahl der Millisekunden nach dem 1. Januar 1970, 00:00:00 GMT zurück. Der Wert ist negativ, wenn der Zeitpunkt vor dem 1.1.1970 liegt.
- `void setTime(long time)`
  Setzt wie der Konstruktor die Anzahl der Millisekunden des Datum-Objekts neu.
- `boolean before(Date when)`
- `boolean after(Date when)`
  Testet, ob das eigene Datum vor oder nach dem übergebenen Datum liegt: Gibt `true` zurück, wenn `when` vor oder nach dem eigenen Datum liegt, sonst `false`. Falls die Millisekunden in `long` bekannt sind, kommt ein Vergleich mit den primitiven Werten zum gleichen Ergebnis.
- `boolean equals(Object obj)`
  Testet die Datumsobjekte auf Gleichheit. Gibt `true` zurück, wenn `getTime()` für den eigenen Zeitwert und das Datumsobjekt hinter `obj` den gleichen Wert ergibt und der aktuelle Parameter nicht `null` ist.
- `int compareTo(Date anotherDate)`
  Vergleicht zwei Datum-Objekte und gibt 0 zurück, falls beide die gleiche Zeit repräsentieren. Der Rückgabewert ist kleiner 0, falls das Datum des aufrufenden Exemplars vor dem Datum von `anotherDate` ist, sonst größer 0.
- `int compareTo(Object o)`
  Ist das übergebene Objekt vom Typ `Date`, dann verhält sich die Methode wie `compareTo()`. Andernfalls löst die Methode eine `ClassCastException` aus. Die Methode ist eine Vorgabe aus der Schnittstelle `Comparable`. Mit der Methode lassen sich `Date`-

Objekte in einem Feld über `Arrays.sort(Object[])` oder `Collections.sort()` einfach sortieren.

- `String toString()`
  Gibt eine Repräsentation des Datums aus. Das Format ist nicht landesspezifisch.

11.3.2 Date-Objekte sind nicht immutable

Dass `Date`-Objekte nicht immutable sind, ist sicherlich aus heutiger Sicht eine große Designschwäche. Immer dann, wenn `Date`-Objekte übergeben und zurückgegeben werden sollen, ist eine Kopie des Zustands das Beste, damit nicht später plötzlich ein verteiltes `Date`-Objekt ungewünschte Änderungen an den verschiedensten Stellen provoziert. Am besten sieht es also so aus:

Listing 11.4: com.tutego.insel.date.Person.java, Person

```
class Person
{
  private Date birthday;

  public void setBirthday( Date birthday )
  {
    this.birthday = new Date( birthday.getTime() );
  }

  public Date getBirthday()
  {
    return new Date( birthday.getTime() );
  }
}
```

Hinweis

Eigentlich hat Sun die verändernden Methoden wie `setHours()` oder `setMinutes()`für *deprecated* erklärt. Allerdings blieb eine Methode außen vor: `setTime(long)`, die die Anzahl der Millisekunden seit dem 1.1.1970 neu setzt. In Programmen sollte diese zustandsverändernde Methode vorsichtig eingesetzt und stattdessen die Millisekunden im Konstruktor für ein neues `Date`-Objekt übergeben werden.

11.4 Calendar und GregorianCalendar

Ein Kalender unterteilt die Zeit in Einheiten wie Jahr, Monat, Tag. Der bekannteste Kalender ist der *gregorianische Kalender*, den Papst Gregor XIII. im Jahre 1582 einführte. Vor seiner Einführung war der *julianische Kalender* populär, der auf Julius Cäsar zurückging – daher auch der Name. Er stammt aus dem Jahr 45 vor unserer Zeitrechnung. Der gregorianische und der julianische Kalender sind Sonnenkalender, die den Lauf der Erde um die Sonne als Basis für die Zeiteinteilung nutzen; der Mond spielt keine Rolle. Daneben gibt es Mondkalender wie den islamischen Kalender und die Lunisolarkalender, die Sonne und Mond miteinander verbinden. Zu diesem Typus gehören der chinesische, der griechische und der jüdische Kalender.

Mit Exemplaren vom Typ `Calendar` ist es möglich, Datum und Uhrzeit in den einzelnen Komponenten wie Jahr, Monat, Tag, Stunde, Minute, Sekunde zu setzen und zu erfragen. Da es unterschiedliche Kalendertypen gibt, ist `Calendar` eine abstrakte Basisklasse, und Unterklassen bestimmen, wie konkret eine Abfrage oder Veränderung für ein bestimmtes Kalendersystem aussehen muss. Bisher bringt die Java-Bibliothek mit der Unterklasse `GregorianCalendar` nur eine öffentliche konkrete Implementierung mit, deren Exemplare Daten und Zeitpunkte gemäß dem gregorianischen Kalender verkörpern. In Java 6 ist eine weitere interne Klasse für einen japanischen Kalender hinzugekommen. IBM hat mit *International Components for Unicode for Java* (ICU4J) unter *http://icu.sourceforge.net/* weitere Klassen wie `ChineseCalendar`, `BuddhistCalendar`, `JapaneseCalendar`, `HebrewCalendar` und `IslamicCalendar` freigegeben. Hier findet sich auch einiges zum Thema Ostertage.

11.4.1 Die abstrakte Klasse Calendar

Die Klasse `Calendar` besitzt zum einen Anfrage- und Modifikationsmethoden für konkrete Exemplare und zum anderen statische Fabrikmethoden. Eine einfache statische Methode ist `getInstance()`, um ein benutzbares Objekt zu bekommen.

```
abstract class java.util.Calendar
implements Serializable, Cloneable, Comparable<Calendar>
```

- `static Calendar getInstance()`
  Liefert einen Standard-`Calendar` mit der Standard-Zeitzone und Standard-Lokalisierung zurück.

Neben der parameterlosen Variante von `getInstance()` gibt es drei weitere Varianten, denen ein `TimeZone`-Objekt und `Locale`-Objekt mit übergeben werden kann. Damit kann dann der Kalender auf eine spezielle Zeitzone und einen Landstrich zugeschnitten werden.

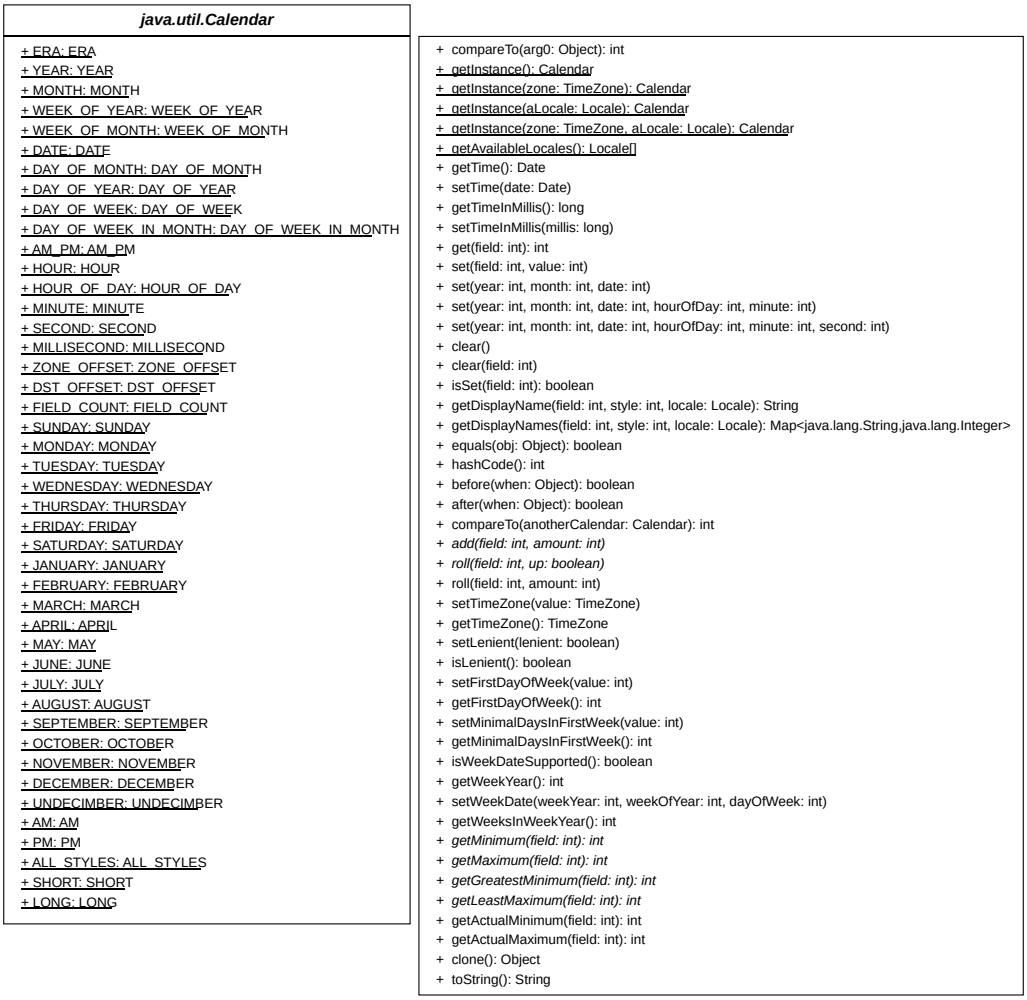

Abbildung 11.2: UML-Diagramm der Klasse Calendar

Hinweis

`Calendar` (bzw. `GregorianCalendar`) hat keine menschenfreundliche `toString()`-Methode. Der String enthält alle Zustände des Objekts:

Hinweis (Forts.)

```
java.util.GregorianCalendar[time=1187732409256,areFieldsSet=true,
areAllFieldsSet=true,lenient=true,zone=sun.util.calendar.ZoneInfo[id="Europe/
Berlin",offset=3600000,dstSavings=3600000,useDaylight=true,transitions=143,
lastRule=java.util.SimpleTimeZone[id=Europe/Berlin,offset=3600000,
dstSavings=3600000,useDaylight=true,startYear=0,startMode=2,startMonth=2,
startDay=-1,startDayOfWeek=1,startTime=3600000,startTimeMode=2,endMode=2,
endMonth=9,endDay=-1,endDayOfWeek=1,endTime=3600000,endTimeMode=2]],
firstDayOfWeek=2,minimalDaysInFirstWeek=4,ERA=1,YEAR=2007,MONTH=7,
WEEK_OF_YEAR=34,WEEK_OF_MONTH=4,DAY_OF_MONTH=21,DAY_OF_YEAR=233,DAY_OF_WEEK=3,
DAY_OF_WEEK_IN_MONTH=3,AM_PM=1,HOUR=11,HOUR_OF_DAY=23,MINUTE=40,SECOND=9,
MILLISECOND=256,ZONE_OFFSET=3600000,DST_OFFSET=3600000]
```


11.4.2 Der gregorianische Kalender

Die Klasse `GregorianCalendar` erweitert die abstrakte Klasse `Calendar`. Sieben Konstruktoren stehen zur Verfügung; vier davon sehen wir uns an:

```
class java.util.GregorianCalendar
extends Calendar
```

- `GregorianCalendar()`
  Erzeugt ein standardmäßiges `GregorianCalendar`-Objekt mit der aktuellen Zeit in der voreingestellten Zeitzone und Lokalisierung.
- `GregorianCalendar(int year, int month, int date)`
  Erzeugt ein `GregorianCalendar`-Objekt in der voreingestellten Zeitzone und Lokalisierung. Jahr, Monat (der zwischen 0 und 11 und nicht zwischen 1 und 12 liegt) und Tag legen das Datum fest.
- `GregorianCalendar(int year, int month, int date, int hour, int minute)`
  Erzeugt ein `GregorianCalendar`-Objekt in der voreingestellten Zeitzone und Lokalisierung. Das Datum legen Jahr, Monat (0 <= `month` <= 11), Tag, Stunde und Minute fest.
- `GregorianCalendar(int year, int month, int date, int hour, int minute, int second)`
  Erzeugt ein `GregorianCalendar`-Objekt in der voreingestellten Zeitzone und Lokalisierung. Das Datum legen Jahr, Monat (0 <= `month` <= 11), Tag, Stunde, Minute und Sekunde fest.

Hinweis

Die Monate beginnen bei 0, sodass `new GregorianCalendar(1973, 3, 12)` nicht den 12. März, sondern den 12. April ergibt! Damit Anfrageprobleme vermieden werden, sollten die `Calendar`-Konstanten `JANUARY` (0), `FEBRUARY`, `MARCH`, `APRIL`, `MAY`, `JUNE`, `JULY`, `AUGUST`, `SEPTEMBER`, `OCTOBER`, `NOVEMBER`, `DECEMBER` (11) verwendet werden. Die spezielle Variable `UNDECIMBER` (12) steht für den dreizehnten Monat, der etwa bei einem Mondkalender anzutreffen ist. Die Konstanten sind keine typsicheren Enums, bieten aber den Vorteil, als `int` einfach mit ihnen zählen zu können.

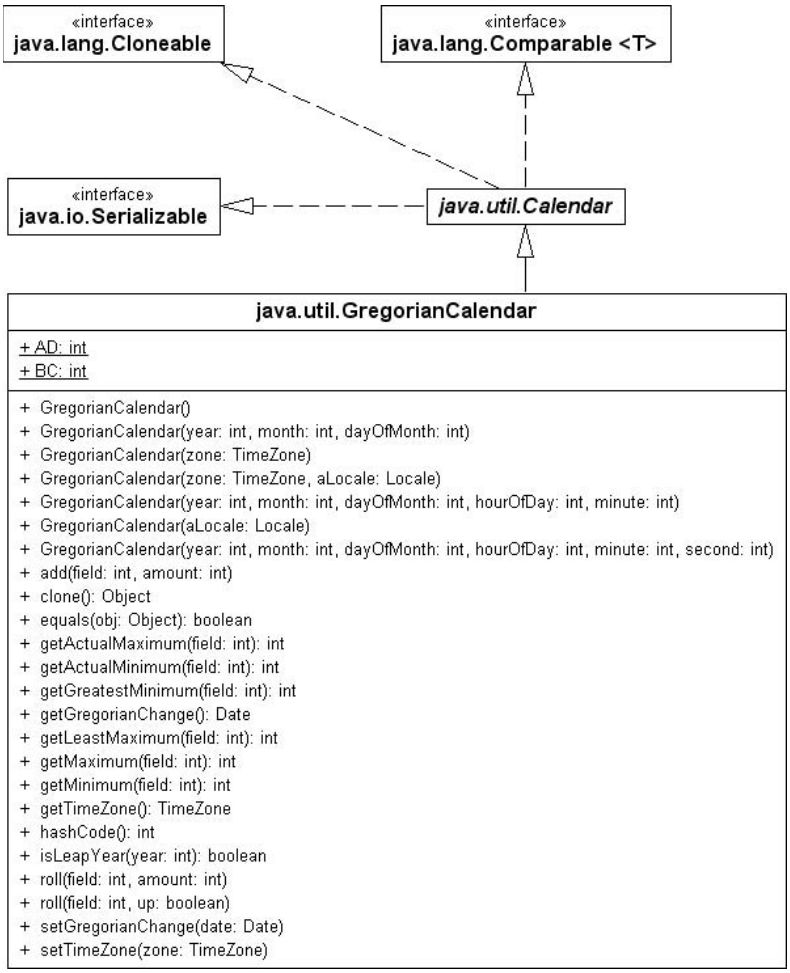

Abbildung 11.3: UML-Diagramm für GregorianCalendar

Neben den hier aufgeführten Konstruktoren gibt es noch weitere, die es erlauben, die Zeitzone und Lokalisierung zu ändern. Standardmäßig eingestellt sind die lokale Zeitzone und die aktuelle Lokalisierung. Ist eines der Argumente im falschen Bereich, löst der Konstruktor eine `IllegalArgumentException` aus.

Hinweis

Zum Aufbau von `Calendar`-Objekten gibt es nun zwei Möglichkeiten:

```
Calendar c = Calendar.getInstance();
```

und

```
Calendar c = new GregorianCalendar();
```

Die erste Variante ist besonders in internationalisierter Software zu bevorzugen, da es einige Länder gibt, die nicht nach dem gregorianischen Kalender arbeiten.

```
Calendar c = Calendar.getInstance( new Locale("ja", "JP", "JP") );
```


11.4.3 Calendar nach Date und Millisekunden fragen

Da `java.util.Date`-Objekte zwar auf den ersten Blick Konstruktoren anbieten, die Jahr, Monat, Tag entgegennehmen, diese Konstruktoren aber veraltet sind, sollten wir den Blick auf `GregorianCalendar` lenken, wie wir das im vorangehenden Abschnitt gemacht haben.

Um von einem `Calendar` die Anzahl der vergangenen Millisekunden seit dem 1.1.1970 abzufragen, dient `getTimeInMillis()` (eine ähnliche Methode hat auch `Date`, nur heißt sie dort `getTime()`).

Beispiel

Bestimme die Anzahl der Tage, die seit einem bestimmten Tag, Monat und Jahr vergangen sind:

```
int date  = 1;
int month = Calendar.JANUARY;
int year  = 1900;
long ms = new GregorianCalendar( year, month, date ).getTimeInMillis();
long days = TimeUnit.MILLISECONDS.toDays( System.currentTimeMillis() - ms );
System.out.println( days );                    // 40303
```

Hinweis

`Calendar` und `Date` haben beide eine `getTime()`-Methode. Nur liefert die `Calendar`-Methode `getTime()` ein `java.util.Date`-Objekt und die `Date`-Methode `getTime()` ein `long`. Gutes API-Design sieht anders aus. Damit Entwickler aber keine unschönen `cal.getTime().getTime()`-Ausdrücke schreiben müssen, um vom `Calendar` die Anzahl der Millisekunden zu beziehen, ist `getTimeInMillis()` im Angebot.

```
abstract class java.util.Calendar
implements Serializable, Cloneable, Comparable<Calendar>
```

- `final long getTimeInMillis()`
  Liefert die seit der Epoche (January 1, 1970 00:00:00.000 GMT, Gregorian) vergangene Zeit in Millisekunden.
- `final Date getTime()`
  Liefert ein `Date`-Objekt zu diesem `Calendar`.

11.4.4 Abfragen und Setzen von Datumselementen über Feldbezeichner

Das Abfragen und Setzen von Datumselementen des gregorianischen Kalenders erfolgt mit den überladenen Methoden `get()` und `set()`. Beide erwarten als erstes Argument einen Feldbezeichner – eine Konstante aus der Klasse `Calendar` –, der angibt, auf welches Datum-/Zeitfeld zugegriffen werden soll. Die `get()`-Methode liefert den Inhalt des angegebenen Felds, und `set()` schreibt den als zweites Argument übergebenen Wert in das Feld.

zB Beispiel

Führe Anweisungen aus, wenn es 19 Uhr ist:

```
if ( Calendar.getInstance().get( Calendar.HOUR_OF_DAY ) == 19 )
 ...
```

Die folgende Tabelle gibt eine Übersicht der Feldbezeichner und ihrer Wertebereiche im Fall des konkreten `GregorianCalendar`.

```
abstract class java.util.Calendar
implements Serializable, Cloneable, Comparable<Calendar>
```

Feldbezeichner Calendar.*	Minimalwert	Maximalwert	Erklärung
ERA	0 (BC)	1 (AD)	Datum vor oder nach Christus
YEAR	1	theoretisch unbeschränkt	Jahr
MONTH	0	11	Monat (nicht von 1 bis 12!)
DAY_OF_MONTH alternativ DATE	1	31	Tag
WEEK_OF_YEAR	1	54	Woche
WEEK_OF_MONTH	1	6	Woche des Monats
DAY_OF_YEAR	1	366	Tag des Jahres
DAY_OF_WEEK	1	7	Tag der Woche (1 = Sonntag, 7 = Samstag)
DAY_OF_WEEK_IN_MONTH	1	6	Tag der Woche im Monat
HOUR	0	11	Stunde von 12
HOUR_OF_DAY	0	23	Stunde von 24
MINUTE	0	59	Minute
SECOND	0	59	Sekunden
MILLISECOND	0	999	Millisekunden
AM_PM	0	1	vor 12, nach 12
ZONE_OFFSET	13*60*60*1000	+14*60*60*1000	Zeitzonenabweichung in Millisekunden
DST_OFFSET	0	2*60*60*1000	Sommerzeitabweichung in Millisekunden

Tabelle 11.2: Konstanten aus der Klasse Calendar

Nun können wir mit den Varianten von `set()` die Felder setzen und mit `get()` wieder hereinholen. Beachtenswert sind der Anfang der Monate mit 0 und der Anfang der Wochentage mit 1 (`SUNDAY`), 2 (`MONDAY`), ..., 7 (`SATURDAY`) – Konstanten der Klasse `Calendar` stehen in Klammern. Die Woche beginnt in der Java-Welt also bei 1 und Sonntag, statt – wie vielleicht anzunehmen – bei 0 und Montag.

zB **Beispiel**

Ist ein `Date`-Objekt gegeben, so speichert es Datum und Zeit. Soll der Zeitanteil gelöscht werden, so bietet Java dafür keine eigene Methode. Die Lösung ist, Stunden, Minuten, Sekunden und Millisekunden von Hand auf 0 zu setzen. Löschen wir vom Hier und Jetzt die Zeit:

```
Date date = new Date();
Calendar cal = Calendar.getInstance();
cal.setTime( date );
cal.set( Calendar.HOUR_OF_DAY, 0 );
cal.set( Calendar.MINUTE, 0 );
cal.set( Calendar.SECOND, 0 );
cal.set( Calendar.MILLISECOND, 0 );
date = cal.getTime();
```

Eine Alternative wäre, den Konstruktor `GregorianCalendar(int year, int month, int dayOfMonth)` mit den Werten vom Datum zu nutzen.

```
abstract class java.util.Calendar
implements Serializable, Cloneable, Comparable<Calendar>
```

- `int get(int field)`
  Liefert den Wert für `field`.
- `void set(int field, int value)`
  Setzt das Feld `field` **mit dem Wert** `value`.
- `final void set(int year, int month, int date)`
  Setzt die Werte für Jahr, Monat und Tag.
- `final void set(int year, int month, int date, int hourOfDay, int minute)`
  Setzt die Werte für Jahr, Monat, Tag, Stunde und Minute.
- `final void set(int year, int month, int date, int hourOfDay, int minute, int second)`
  Setzt die Werte für Jahr, Monat, Tag, Stunde, Minute und Sekunde.

➔ **Hinweis**

Wo die `Date`-Klasse etwa spezielle (veraltete) Methoden wie `getYear()`, `getDay()`, `getHours()` anbietet, so müssen Nutzer der `Calendar`-Klasse immer die `get(field)`-Methode nutzen. Es gibt keinen Getter für den Zugriff auf ein bestimmtes Feld.

Werte relativ setzen

Neben der Möglichkeit, die Werte entweder über den Konstruktor oder über `set()` absolut zu setzen, sind auch relative Veränderungen möglich. Dazu wird die `add()`-Methode eingesetzt, die wie `set()` als erstes Argument einen Feldbezeichner bekommt und als zweites die Verschiebung.

Beispiel zB

Was ist der erste und letzte Tag einer Kalenderwoche?

```
Calendar cal = Calendar.getInstance();
cal.set( Calendar.WEEK_OF_YEAR, 15 );
cal.set( Calendar.DAY_OF_WEEK, Calendar.MONDAY );
System.out.printf( "%tD ", cal );                // 04/09/07
cal.add( Calendar.DAY_OF_WEEK, 6 );
System.out.printf( "%tD", cal );                 // 04/15/07
```

Die Methode `add()` setzt das Datum um sechs Tage hoch.

Da es keine `sub()`-Methode gibt, können die Werte bei `add()` auch negativ sein.

Beispiel zB

Wo waren wir heute vor einem Jahr?

```
Calendar cal = Calendar.getInstance();
System.out.printf( "%tF%n", cal );              // 2006-06-09
cal.add( Calendar.YEAR, -1 );
System.out.printf( "%tF%n", cal );              // 2005-06-09
```

Eine weitere Methode `roll()` ändert keine folgenden Felder, was `add()` macht, wenn etwa zum Dreißigsten eines Monats zehn Tage addiert werden.

```
abstract class java.util.Calendar
implements Serializable, Cloneable, Comparable<Calendar>
```

- `abstract void add(int field, int amount)`
  Addiert (bzw. subtrahiert, wenn `amount` negativ ist) den angegeben Wert auf dem (bzw. vom) Feld.

- `abstract void roll(int field, boolean up)`
  Setzt eine Einheit auf dem gegebenen Feld hoch oder runter, ohne die nachfolgenden Felder zu beeinflussen. Ist der aktuelle Feldwert das Maximum (bzw. Minimum) und wird um eine Einheit addiert (bzw. subtrahiert), ist der nächste Feldwert das Minimum (bzw. Maximum).
- `void roll(int field, int amount)`
  Ist `amount` positiv, führt diese Methode die Operation `roll(field, true)` genau `amount`-mal aus, ist `amount` negativ, dann wird `amount`-mal `roll(field, false)` aufgerufen.

In `GregorianCalendar` ist die Implementierung in Wirklichkeit etwas anders. Da ist `roll(int, int)` implementiert, und `roll(int, boolean)` ist ein Aufruf von `roll(field, up ? +1 : -1)`.

11.5 Klassenlader (Class Loader)

Ein Klassenlader ist dafür verantwortlich, eine Klasse zu laden. Aus der Datenquelle (im Allgemeinen einer Datei) liefert der Klassenlader ein Byte-Feld mit den Informationen, die im zweiten Schritt dazu verwendet werden, die Klasse im Laufzeitsystem einzubringen; das ist *Linking*. Es gibt eine Reihe von vordefinierten Klassenladern und die Möglichkeit, eigene Klassenlader zu schreiben, um etwa verschlüsselte und komprimierte *.class*-Dateien zu laden.

11.5.1 Woher die kleinen Klassen kommen

Nehmen wir zu Beginn ein einfaches Programm mit zwei Klassen:

```
class A
{
  static String s = new java.util.Date().toString();

  public static void main( String[] args )
  {
    B b = new B();
  }
}

class B
```

```
{
  A a;
}
```

Wenn die Laufzeitumgebung das Programm `A` startet, muss sie eine Reihe von Klassen laden. Sofort wird klar, dass es zumindest `A` sein muss. Wenn aber die statische `main()`-Methode aufgerufen wird, muss auch `B` geladen sein. Und da beim Laden einer Klasse auch die statischen Variablen initialisiert werden, wird auch die Klasse `java.util.Date` geladen. Zwei weitere Dinge werden nach einiger Überlegung deutlich:

- Wenn `B` geladen wird, bezieht es sich auf `A`. Da `A` aber schon geladen ist, muss es nicht noch einmal geladen werden.
- Unsichtbar stecken noch andere referenzierte Klassen dahinter, die nicht direkt sichtbar sind. So wird zum Beispiel `Object` geladen werden, da implizit in der Klassendeklaration von `A` steht: `class A extends Object`.

Im Beispiel mit den Klassen `A` und `B` lädt die Laufzeitumgebung selbstständig die Klassen (*implizites Klassenladen*). Klassen lassen sich auch mit `Class.forName()` über ihren Namen laden (*explizites Klassenladen)*.

Hinweis

Um zu sehen, welche Klassen überhaupt geladen werden, lässt sich der virtuellen Maschine beim Start der Laufzeitumgebung ein Schalter mitgeben – `verbose:class`. Dann gibt die Maschine beim Lauf alle Klassen aus, die sie lädt.

Die Suchorte

Ein festes, dreistufiges Schema bestimmt die Suche nach den Klassen:

1. Klassen wie `String`, `Object` oder `Point` stehen in einem ganz speziellen Archiv. Wenn ein eigenes Java-Programm gestartet wird, so sucht die virtuelle Maschine die angeforderten Klassen zuerst in diesem Archiv. Da es elementare Klassen sind, die zum Hochfahren eines Systems gehören, werden sie *Bootstrap-Klassen* genannt. Das Archiv mit diesen Klassen heißt oft *rt.jar* (für Runtime). Andere Archive können hinzukommen – wie *i18n.jar*, das Internationalisierungsdaten beinhaltet. Die Implementierung dieses Bootstrap-Laders ist nicht öffentlich und wird von System zu System unterschiedlich sein.
2. Findet die Laufzeitumgebung die Klassen nicht bei den Bootstrap-Klassen, so werden alle Archive eines speziellen Verzeichnisses untersucht, das sich *Extension-Verzeich-*

nis nennt. Das Verzeichnis gibt es bei jeder Java-Version. Es liegt unter *lib/ext*. Werden hier Klassen eingelagert, so findet die Laufzeitumgebung diese Klassen ohne weitere Anpassung und Setzen von Pfaden. In sonstige Verzeichnisse einer Java-Installation sollten keine Klassen kopiert werden.

3. Ist eine Klasse auch im Erweiterungsverzeichnis nicht zu finden, beginnt die Suche im *Klassenpfad* (*Classpath*). Diese Pfadangabe besteht aus einer Aufzählung einzelner Verzeichnisse, Klassen oder Jar-Archive, in denen die Laufzeitumgebung nach den Klassendateien sucht. Standardmäßig ist dieser Klassenpfad auf das aktuelle Verzeichnis gesetzt (».«).

Hinweis

Es gibt spezielle Bootstrap-Klassen, die sich überschreiben lassen. Sie werden in das spezielle Verzeichnis *endorsed* gesetzt. Mehr Informationen dazu folgen in Abschnitt 11.5.6.

11.5.2 Setzen des Klassenpfades

Die Suchorte lassen sich angeben, wobei die Bestimmung des Klassenpfades für die eigenen Klassen die wichtigste ist. Sollen in einem Java-Projekt Dateien aus einem Verzeichnis oder einem externen Java-Archiv geholt werden, so ist der übliche Weg, dieses Verzeichnis oder Archiv im Klassenpfad anzugeben. Dafür gibt es zwei Varianten. Die erste ist, über den Schalter `-classpath` (kurz `-cp`) beim Start der virtuellen Maschine die Quellen aufzuführen:

```
$ java -classpath classpath1;classpath2 MainClass
```

Eine Alternative ist das Setzen der Umgebungsvariablen `CLASSPATH` mit einer Zeichenfolge, die die Klassen spezifiziert:

```
$ SET CLASSPATH=classpath1;classpath2
$ java MainClass
```

Ob der Klassenpfad überhaupt gesetzt ist, ermittelt ein einfaches `echo $CLASSPATH` (Unix) beziehungsweise `echo %CLASSPATH%` (Windows).

Hinweis

Früher – das heißt vor Java 1.2 – umfasste der `CLASSPATH` auch die Bootstrap-Klassen. Das ist seit 1.2 überflüssig und bedeutet: Die typischen Klassen aus den Paketen `java.*`, `com.sun.*` usw. wie `String` stehen nicht im `CLASSPATH`.

Zur Laufzeit steht dieser Klassenpfad in der Umgebungsvariablen `java.class.path`. Auch die Bootstrap-Klassen können angegeben werden. Dazu dient der Schalter `-Xbootclasspath` oder die Variable `sun.boot.class.path`. Zusätzliche Erweiterungsverzeichnisse lassen sich über die Systemeigenschaft `java.ext.dirs` zuweisen.

Listing 11.5: Ausgaben von com/tutego/insel/lang/ClasspathDir.java

Eigenschaft	Beispielbelegung
`java.class.path`	*C:\Insel\programme\1_11_Java_Library\bin*
`java.ext.dirs`	*C:\Program Files\Java\jdk1.7.0\jre\lib\ext;* *C:\Windows\Sun\Java\lib\ext*
`sun.boot.class.path`	*C:\Program Files\Java\jdk1.7.0\jre\lib\resources.jar;* *C:\Program Files\Java\jdk1.7.0\jre\lib\rt.jar;* *C:\Program Files\Java\jdk1.7.0\jre\lib\sunrsasign.jar;* *C:\Program Files\Java\jdk1.7.0\jre\lib\jsse.jar;* *C:\Program Files\Java\jdk1.7.0\jre\lib\jce.jar;* *C:\Program Files\Java\jdk1.7.0\jre\lib\charsets.jar;* *C:\Program Files\Java\jdk1.7.0\jre\lib\modules\jdk.boot.jar;* *C:\Program Files\Java\jdk1.7.0\jre\classes*

Tabelle 11.3: Beispielbelegungen der Variablen

Hinweis

Wird die JVM über `java -jar` aufgerufen, beachtet sie nur Klassen in dem genannten Jar und ignoriert den Klassenpfad.

11.5.3 Die wichtigsten drei Typen von Klassenladern

Eine Klassendatei kann von der Java-Laufzeitumgebung über verschiedene Klassenlader bezogen werden. Die wichtigsten sind: Bootstrap-, Erweiterungs- und Applikations-Klassenlader. Sie arbeiten insofern zusammen, als dass sie sich gegenseitig Aufgaben zuschieben, wenn eine Klasse nicht gefunden wird:

- *Bootstrap-Klassenlader*: für die Bootstrap-Klassen
- *Erweiterungs-Klassenlader*: für die Klassen im *lib/ext*-Verzeichnis
- *Applikations-Klassenlader* (auch *System-Klassenlader*): Der letzte Klassenlader im Bunde berücksichtigt bei der Suche den `java.class.path`.

Aus Sicherheitsgründen beginnt der Klassenlader bei einer neuen Klasse immer mit dem System-Klassenlader und reicht dann die Anfrage weiter, wenn er selbst die Klasse nicht laden konnte. Dazu sind die Klassenlader miteinander verbunden. Jeder Klassenlader `L` hat dazu einen Vater-Klassenlader `V`. Erst darf der Vater versuchen, die Klassen zu laden. Kann er es nicht, gibt er die Arbeit an `L` ab.

Hinter dem letzten Klassenlader können wir einen eigenen *benutzerdefinierten Klassenlader* installieren. Auch dieser wird einen Vater haben, den üblicherweise der Applikations-Klassenlader verkörpert.

11.5.4 Die Klasse java.lang.ClassLoader *

Jeder Klassenlader in Java ist vom Typ `java.lang.ClassLoader`. Die Methode `loadClass()` erwartet einen sogenannten »binären Namen«, der an den vollqualifizierten Klassennamen erinnert.

```
abstract class java.lang.ClassLoader
```

- `protected Class<?> loadClass(String name, boolean resolve)`
  Lädt die Klasse und bindet sie mit `resolveClass()` ein, wenn `resolve` gleich `true` ist.
- `public Class<?> loadClass(String name)`
  Die öffentliche Methode ruft `loadClass(name, false)` auf, was bedeutet, dass die Klasse nicht standardmäßig angemeldet (gelinkt) wird. Beide Methoden können eine `ClassNotFoundException` auslösen.

Die geschützte Methode führt anschließend drei Schritte durch:

1. Wird `loadClass()` auf einer Klasse aufgerufen, die dieser Klassenlader schon eingelesen hat, so kehrt die Methode mit dieser gecachten Klasse zurück.
2. Ist die Klasse nicht gespeichert, darf zuerst der Vater (*parent class loader*) versuchen, die Klasse zu laden.
3. Findet der Vater die Klasse nicht, so darf jetzt der Klassenlader selbst mit `findClass()` versuchen, die Klasse zu beziehen.

Eigene Klassenlader überschreiben in der Regel die Methode `findClass()`, um nach einem bestimmten Schema zu suchen, etwa nach Klassen aus der Datenbank. In diesen Stufen ist es auch möglich, höher stehende Klassenlader zu umgehen, was beispielsweise bei Servlets Anwendung findet.

Neue Klassenlader

Java nutzt an den verschiedensten Stellen spezielle Klassenlader, etwa für Applets den `sun.applet.AppletClassLoader`. Für uns ist der `java.net.URLClassLoader` interessant, da er Klassen von beliebigen URLs laden kann und die Klassen nicht im klassischen Klassenpfad benötigt. Wie ein eigener Klassenlader aussieht, zeigt das Beispiel unter *http://tutego.com/go/urlclassloader*, das den URL-Classloader vom Prinzip her nachimplementiert.

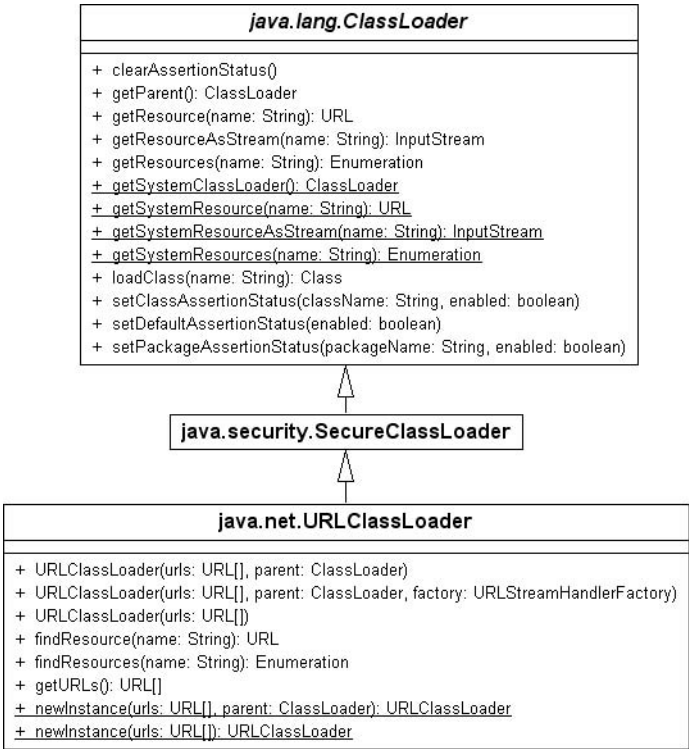

Abbildung 11.4: Klassenhierarchie von URLClassLoader

11.5.5 Hot Deployment mit dem URL-Classloader *

Unter *Hot Deployment* ist die Möglichkeit zu verstehen, zur Laufzeit Klassen auszutauschen. Diese Möglichkeit ist für viele EJB- oder Servlet-Container wichtig, da sie im Dateisystem auf eine neue Klassendatei warten und im gegebenen Fall die alte Klasse durch eine neue ersetzen. Ein Servlet-Container überwacht geladene Klassen und lädt sie bei Änderungen neu. Eine Internetsuche mit dem Stichwort *AdaptiveClassLoader* listet Implementierungen auf.

Damit dieser heiße Wechsel funktioniert, muss die Klasse über einen neuen Klassenlader bezogen werden. Das liegt daran, dass der Standardklassenlader von Haus aus keine Klasse mehr loswird, wenn er sie einmal geladen hat. Mit anderen Worten: Wenn eine Klasse über `Class.forName(Klasse)` angefordert wird, ist sie immer im Cache und wird nicht mehr entladen. Ein neuer Klassenlader fängt immer von vorn an, wenn er die Klasse für sich zum ersten Mal sieht.

Mit immer neuen Klassenladern funktioniert das Neuladen, weil für eine neue Klasse dann jeweils ein eigener Klassenlader zuständig ist. Ändert sich die Klasse, wird ein neuer Klassenlader konstruiert, der die neue Klasse lädt. Doch damit ist die alte Klasse noch nicht aus dem Spiel. Nur wenn sich niemand mehr für die alte Klasse und für den Klassenlader interessiert, kann die Laufzeitumgebung diese nicht benutzten Objekte erkennen und aufräumen.

Gleiche Klasse mehrfach laden

Wir wollen im Folgenden eine eigene statische Methode `newInstance()` vorstellen, die beim Aufruf die neueste Version des Dateisystems lädt und ein Exemplar bildet. Die neu zu ladende Klasse soll ohne Beschränkung der Allgemeinheit einen Standard-Konstruktor haben – andernfalls muss über Reflection ein parametrisierter Konstruktor aufgerufen werden; wir wollen das Beispiel aber kurz halten.

Beginnen wir mit der Klasse, die zweimal geladen werden soll. Sie besitzt einen statischen Initialisierungsblock, der etwas auf der Konsole ausgibt, wenn er beim Laden ausgeführt wird:

Listing 11.6: com/tutego/insel/lang/ClassToLoadMultipleTimes.java

```
package com.tutego.insel.lang;

public class ClassToLoadMultipleTimes
{
  static
  {
    System.out.println( "ClassToLoadMultipleTimes" );
  }
}
```

Die Testklasse legen wir unter *C:* ab, und zwar so, dass die Verzeichnisstruktur durch das Paket erhalten bleibt – demnach unter *C:\com\tutego\insel\lang*.

Jetzt brauchen wir noch eine Testklasse, die `ClassToLoadMultipleTimes` unter dem Wurzelverzeichnis liest (also etwa unter *C:/*):

Listing 11.7: com/tutego/insel/lang/LoadClassMultipleTimes.java

```
package com.tutego.insel.lang;

import java.io.File;
import java.net.*;

public class LoadClassMultipleTimes
{
  static Object newInstance( String path, String classname ) throws Exception
  {
    URL url = new File( path ).toURI().toURL();

    URLClassLoader cl = new URLClassLoader( new URL[]{ url } );

    Class<?> c = cl.loadClass( classname );

    return c.newInstance();
  }

  public static void main( String[] args ) throws Exception
  {
    newInstance( "/", "com.tutego.insel.lang.ClassToLoadMultipleTimes" );
    newInstance( "/", "com.tutego.insel.lang.ClassToLoadMultipleTimes" );
  }
}
```

Nach dem direkten Start ohne Vorbereitung bekommen wir nur einmal die Ausgabe – anders als erwartet. Der Grund liegt in der Hierarchie der Klassenlader. Wichtig ist hier, dass der Standardklassenlader die Klasse `ClassToLoadMultipleTimes` nicht »sehen« darf. Wir müssen die Klasse also aus dem Zugriffspfad der Laufzeitumgebung löschen, da andernfalls aufgrund des niedrigen Rangs unser eigener URL-Klassenlader nicht zum Zuge kommt. (Und ist die Klassendatei nicht im Pfad, können wir das praktische `ClassToLoadMultipleTimes.class.getName()` nicht nutzen.) Erst nach dem Löschen werden wir Zeuge,

wie die virtuelle Maschine auf der Konsole die beiden Meldungen ausgibt, wenn der statische Initialisierungsblock ausgeführt wird.

Die zu ladende Klasse darf nicht den gleichen voll qualifizierten Namen wie eine Standardklasse (etwa `java.lang.String`) tragen. Das liegt daran, dass auch in dem Fall, in dem die Klasse mit dem eigenen `URLClassLoader` bezogen werden soll, die Anfrage trotzdem erst an den System-Klassenlader, dann an den Erweiterungs-Klassenlader und erst ganz zum Schluss an unseren eigenen Klassenlader geht. Es ist also nicht möglich, aus einem Java-Programm Klassen zu beziehen, die prinzipiell vom System-Klassenlader geladen werden. Wir können eine Klasse wie `javax.swing.JButton` nicht selbst beziehen. Wenn sie mit einem Klassenlader ungleich unserem eigenen geladen wird, hat dies wiederum zur Folge, dass wir die geladene Klasse nicht mehr loswerden – was allerdings im Fall der Systemklassen kein Problem sein sollte.

Implementiert die Klasse eine bestimmte Schnittstelle oder erbt sie von einer Basisklasse, lässt sich der Typ der Rückgabe unserer Methode `newInstance()` einschränken. Auf diese Weise ist ein Plugin-Prinzip realisierbar: Die geladene Klasse bietet mit dem Typ Methoden an. Während dieser Typ bekannt ist (der implizite Klassenlader besorgt sie), wird die Klasse selbst erst zur Laufzeit geladen (expliziter Klassenlader).

! **Einzigartigkeit eines Singletons**

Ein Singleton ist ein Erzeugermuster, das ein Exemplar nur einmal hervorbringt. Singletons finden sich in der JVM an einigen Stellen; so gibt es `java.lang.Runtime` nur einmal, genauso wie `java.awt.Toolkit`. Auch Enums sind Singletons, und so lassen sich die Aufzählungen problemlos mit `==` vergleichen. Und doch gibt es zwischen den Bibliotheks-Singletons und den von Hand gebauten Singleton-Realisierungen und Enums einen großen Unterschied: Sie basieren alle auf statischen Variablen, die dieses eine Exemplar referenzieren. Damit ist eine Schwierigkeit verbunden. Denn wie wir an den Beispielen mit dem `URLClassLoader` gesehen haben, ist dieses Exemplar immer nur pro Klassenlader einzigartig, aber nicht in der gesamten JVM an sich, die eine unbestimmte Anzahl von Klassenladern nutzen kann. Die Enums sind ein gutes Beispiel. In einem Server kann es zwei gleiche `Weekday`-Aufzählungen im gleichen Paket geben. Und doch sind sie völlig unterschiedlich und miteinander inkompatibel, wenn sie zwei unterschiedliche Klassenlader einlesen. Selbst die `Class`-Objekte dieser Enums, die ja auch Singletons innerhalb eines Klassenladers sind, sind bei zwei verschiedenen Klassenladern nicht identisch. Globale Singletons für die gesamte JVM gibt es nicht – zum Glück. Auf der anderen Seite verursachen diese Klassen-Phantome viele Probleme in Java EE-Umgebungen. Doch das ist eine andere Geschichte für ein Java EE-Buch.

```
class java.net.URLClassLoader
extends SecureClassLoader
```

- `URLClassLoader(URL[] urls)`
  Erzeugt einen neuen `URLClassLoader` für ein Feld von URLs mit dem Standard-Vater-Klassenlader.
- `URLClassLoader(URL[] urls, ClassLoader parent)`
  Erzeugt einen neuen `URLClassLoader` für ein Feld von URLs mit einem gegebenen Vater-Klassenlader.
- `protected void addURL(URL url)`
  Fügt eine URL hinzu.
- `URL[] getURLs()`
  Liefert die URLs.

11.5.6 Das Verzeichnis jre/lib/endorsed *

Im Fall der XML-Parser und weiterer Bibliotheken kommt es häufiger vor, dass sich die Versionen einmal ändern. Es wäre nun müßig, aus diesem Grund die neuen Bibliotheken immer im `bootclasspath` aufzunehmen, da dann immer eine Einstellung über die Kommandozeile stattfände. Die Entwickler haben daher für spezielle Pakete ein Verzeichnis vorgesehen, in dem Updates eingelagert werden können: das Verzeichnis *jre/lib/endorsed* der Java-Installation. Alternativ können die Klassen und Archive auch durch die Kommandozeilenoption `java.endorsed.dirs` spezifiziert werden.

Wenn der Klassenlader im Verzeichnis *endorsed* eine neue Version – etwa vom XML-Parser – findet, lädt er die Klassen von dort und nicht aus dem Jar-Archiv, aus dem sonst die Klassen geladen würden. Standardmäßig bezieht er die Ressourcen aus der Datei *rt.jar*. Alle im Verzeichnis *endorsed* angegebenen Typen überdecken somit die Standardklassen aus der Java SE; neue Versionen lassen sich einfach einspielen.

Nicht alle Klassen lassen sich mit *endorsed* überdecken. Zum Beispiel lässt sich keine neue Version von `java.lang.String` einfügen. Die Dokumentation »Endorsed Standards Override Mechanism« unter *http://download.oracle.com/javase/7/docs/technotes/guides/standards/* zeigt die überschreibbaren Pakete an: `javax.rmi.CORBA`, `org.omg.*`, `org.w3c.dom` und `org.xml.*`. (Im Übrigen definiert auch Tomcat, die Servlet-Engine, ein solches Überschreibverzeichnis. Hier können Sie Klassen in das Verzeichnis *common/lib/endorsed* aufnehmen, die dann beim Start von Tomcat die Standardklassen überschreiben.)

11.6 Die Utility-Klasse System und Properties

In der Klasse `java.lang.System` finden sich Methoden zum Erfragen und Ändern von Systemvariablen, zum Umlenken der Standard-Datenströme, zum Ermitteln der aktuellen Zeit, zum Beenden der Applikation und noch für das ein oder andere. Alle Methoden sind ausschließlich statisch, und ein Exemplar von `System` lässt sich nicht anlegen. In der Klasse `java.lang.Runtime` – die Schnittstelle `RunTime` aus dem CORBA-Paket hat hiermit nichts zu tun – finden sich zusätzlich Hilfsmethoden, wie etwa das Starten von externen Programmen oder Methoden zum Erfragen des Speicherbedarfs. Anders als `System` ist hier nur eine Methode statisch, nämlich die Singleton-Methode `getRuntime()`, die das Exemplar von `Runtime` liefert.

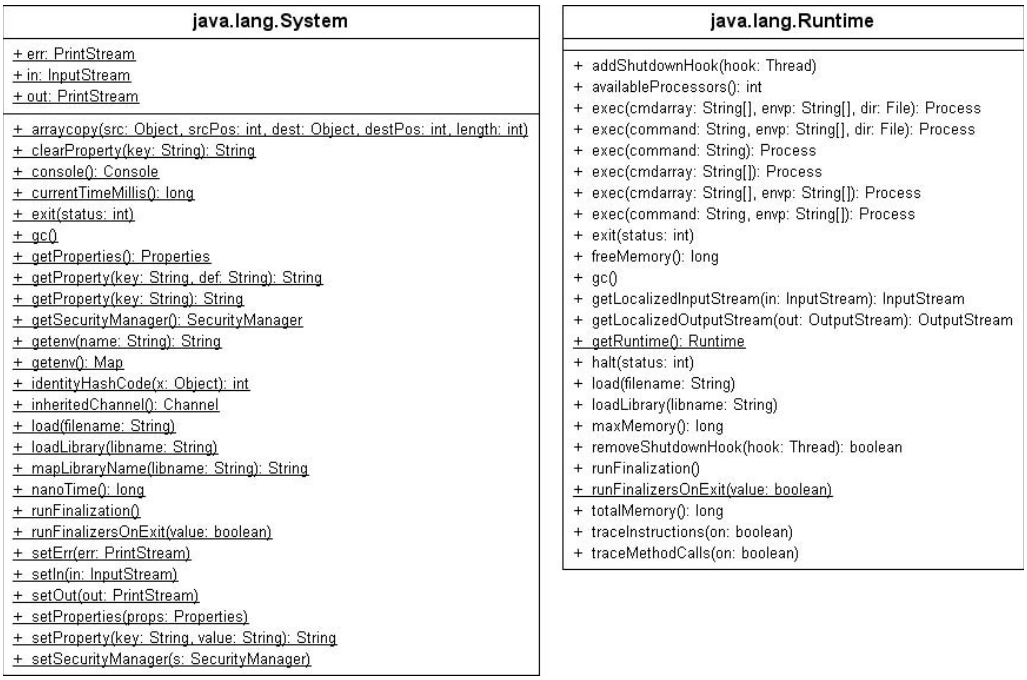

Abbildung 11.5: Eigenschaften der Klassen System und Runtime

> **Bemerkung**
>
> Insgesamt machen die Klassen `System` und `Runtime` keinen besonders aufgeräumten Eindruck; sie wirken irgendwie so, als sei hier alles zu finden, was an anderer Stelle nicht mehr hineingepasst hat. Auch wären Methoden einer Klasse genauso gut in der anderen Klasse aufgehoben.

Bemerkung (Forts.)

Dass die statische Methode `System.arraycopy()` zum Kopieren von Feldern nicht in `java.util.Arrays` stationiert ist, lässt sich nur historisch erklären. Und `System.exit()` leitet an `Runtime.getRuntime().exit()` weiter. Einige Methoden sind veraltet beziehungsweise anders verteilt: Das `exec()` von `Runtime` zum Starten von externen Prozessen übernimmt eine neue Klasse `ProcessBuilder`, und die Frage nach dem Speicherzustand oder der Anzahl der Prozessoren beantworten MBeans, wie etwa `ManagementFactory.getOperatingSystemMXBean().getAvailableProcessors()`. Aber API-Design ist wie Sex: Eine unüberlegte Aktion, und es lebt mit uns für immer.

11.6.1 Systemeigenschaften der Java-Umgebung

Die Java-Umgebung verwaltet Systemeigenschaften wie Pfadtrenner oder die Version der virtuellen Maschine in einem `java.util.Properties`-Objekt. Die statische Methode `System.getProperties()` erfragt diese Systemeigenschaften und liefert das gefüllte `Properties`-Objekt zurück. Zum Erfragen einzelner Eigenschaften ist das `Properties`-Objekt aber nicht unbedingt nötig: `System.getProperty()` erfragt direkt eine Eigenschaft.

Beispiel

Gib den Namen des Betriebssystems aus:

```
System.out.println( System.getProperty("os.name") );
```

Gib alle Systemeigenschaften auf dem Bildschirm aus:

```
System.getProperties().list( System.out );
```

Die Ausgabe beginnt mit:

```
-- listing properties --
java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=C:\Program Files\Java\jdk1.7.0\jre\bin
java.vm.version=21.0-b17
java.vm.vendor=Oracle Corporation
java.vendor.url=http://java.oracle.com/
path.separator=;
```

Eine Liste der wichtigen Standard-Systemeigenschaften:

Schlüssel	Bedeutung
java.version	Version der Java-Laufzeitumgebung
java.class.path	Klassenpfad
java.library.path	Pfad für native Bibliotheken
java.io.tmpdir	Pfad für temporäre Dateien
os.name	Name des Betriebssystems
file.separator	Trenner der Pfadsegmente, etwa / (Unix) oder \ (Windows)
path.separator	Trenner bei Pfadangaben, etwa : (Unix) oder ; (Windows)
line.separator	Zeilenumbruchzeichen(-folge)
user.name	Name des angemeldeten Benutzers
user.home	Home-Verzeichnis des Benutzers
user.dir	Aktuelles Verzeichnis des Benutzers

Tabelle 11.4: Standardsystemeigenschaften

API-Dokumentation

Ein paar weitere Schlüssel zählt die API-Dokumentation bei `System.getProperties()` auf. Einige der Variablen sind auch anders zugänglich, etwa über die Klasse `File`.

```
final class java.lang.System
```

- `static String getProperty(String key)`
  Gibt die Belegung einer Systemeigenschaft zurück. Ist der Schlüssel `null` oder leer, gibt es eine `NullPointerException` beziehungsweise eine `IllegalArgumentException`.
- `static String getProperty(String key, String def)`
  Gibt die Belegung einer Systemeigenschaft zurück. Ist sie nicht vorhanden, liefert die Methode die Zeichenkette `def`, den Default-Wert. Für die Ausnahmen gilt das Gleiche wie bei `getProperty(String)`.
- `static String setProperty(String key, String value)`
  Belegt eine Systemeigenschaft neu. Die Rückgabe ist die alte Belegung – oder `null`, falls es keine alte Belegung gab.

- `static String clearProperty(String key)`
  Löscht eine Systemeigenschaft aus der Liste. Die Rückgabe ist die alte Belegung – oder `null`, falls es keine alte Belegung gab.
- `static Properties getProperties()`
  Liefert ein mit den aktuellen Systembelegungen gefülltes `Properties`-Objekt.

11.6.2 line.separator

Um nach dem Ende einer Zeile an den Anfang der nächsten zu gelangen, wird ein *Zeilenumbruch* (engl. *new line*) eingefügt. Das Zeichen für den Zeilenumbruch muss kein einzelnes sein, es können auch mehrere Zeichen nötig sein. Zum Leidwesen der Programmierer unterscheidet sich die Anzahl der Zeichen für den Zeilenumbruch auf den bekannten Architekturen:

- Unix: Line Feed (Zeilenvorschub)
- Windows: beide Zeichen (Carriage Return und Line Feed)
- Macintosh: Carriage Return (Wagenrücklauf)

Der Steuercode für Carriage Return (kurz CR) ist 13 (0x0D), der für Line Feed (kurz LF) 10 (0x0A). Java vergibt obendrein eigene Escape-Sequenzen für diese Zeichen: `\r` für Carriage Return und `\n` für Line Feed (die Sequenz `\f` für einen Form Feed – Seitenvorschub – spielt bei den Zeilenumbrüchen keine Rolle).

Bei der Ausgabe mit einem `println()` oder der Nutzung des Formatspezifizierers `%n` in `format()` beziehungsweise `printf()` haben wir bei Zeilenumbrüchen keinerlei Probleme. So ist es oft gar nicht nötig, das Zeilenumbruchzeichen vom System über die Property `line.separator` zu erfragen.

11.6.3 Eigene Properties von der Konsole aus setzen *

Eigenschaften lassen sich auch beim Programmstart von der Konsole aus setzen. Dies ist praktisch für eine Konfiguration, die beispielsweise das Verhalten des Programms steuert. In der Kommandozeile werden mit `-D` der Name der Eigenschaft und nach einem Gleichheitszeichen (ohne Leerzeichen) ihr Wert angegeben. Das sieht dann etwa so aus:

```
$ java -DLOG -DUSER=Chris -DSIZE=100 com.tutego.insel.lang.SetProperty
```

Die Property LOG ist einfach nur »da«, aber ohne zugewiesenen Wert. Die nächsten beiden Properties, USER und SIZE, sind mit Werten verbunden, die erst einmal vom Typ String sind und vom Programm weiterverarbeitet werden müssen.

Die Informationen tauchen nicht bei der Argumentliste in der statischen main()-Methode auf, da sie vor dem Namen der Klasse stehen und bereits von der Java-Laufzeitumgebung verarbeitet werden.

Um die Eigenschaften auszulesen, nutzen wir das bekannte System.getProperty():

Listing 11.8: com/tutego/insel/lang/SetProperty.java

```
package com.tutego.insel.lang;

class SetProperty
{
  static public void main( String[] args )
  {
    String logProperty      = System.getProperty( "LOG" );
    String usernameProperty = System.getProperty( "USER" );
    String sizeProperty     = System.getProperty( "SIZE" );

    System.out.println( logProperty != null );                    // true

    System.out.println( usernameProperty );                       // Chris

    if ( sizeProperty != null )
      System.out.println( Integer.parseInt( sizeProperty ) );     // 100

    System.out.println( System.getProperty( "DEBUG", "false" ) );  // false
  }
}
```

Wir bekommen über getProperty() einen String zurück, der den Wert anzeigt. Falls es überhaupt keine Eigenschaft dieses Namens gibt, erhalten wir stattdessen null. So wissen wir auch, ob dieser Wert überhaupt gesetzt wurde. Ein einfacher Test wie bei logProperty != null sagt also, ob logProperty vorhanden ist oder nicht. Statt -DLOG führt auch -DLOG= zum gleichen Ergebnis, denn der assoziierte Wert ist der Leerstring. Da alle Properties erst einmal vom Typ String sind, lässt sich usernameProperty einfach ausgeben, und wir bekommen entweder null oder den hinter = angegebenen String. Sind die

Typen keine Strings, müssen sie weiterverarbeitet werden, also etwa mit `Integer.parseInt()`, `Double.parseDouble()` usw. Nützlich ist die Methode `System.getProperty()`, der zwei Argumente übergeben werden, denn das zweite steht für einen Default-Wert. So kann immer ein Standardwert angenommen werden.

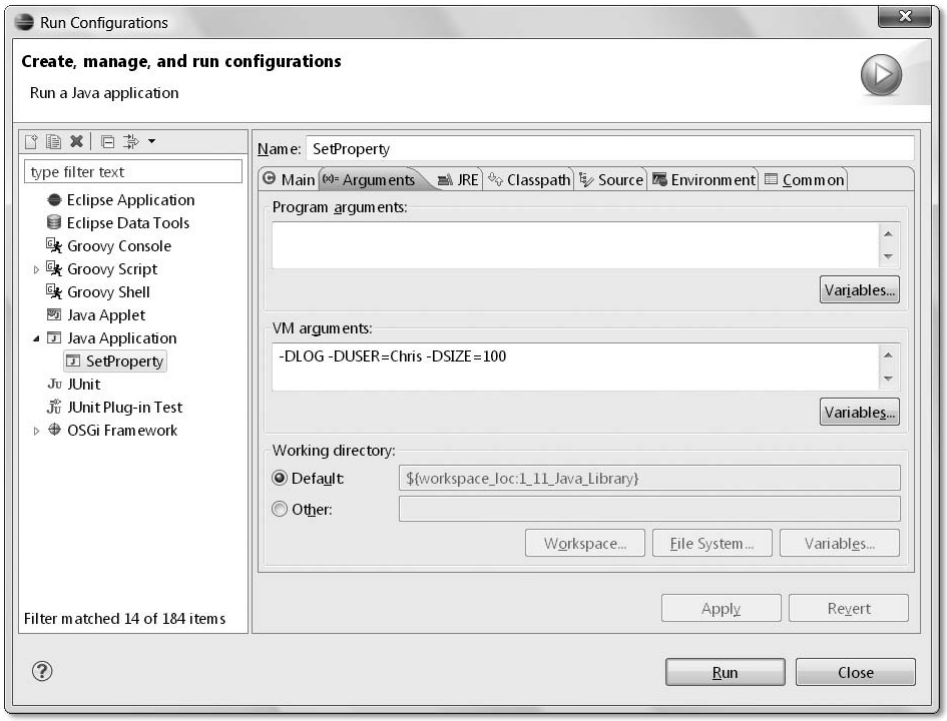

Abbildung 11.6: Entwicklungsumgebungen erlauben es, die Kommandozeilenargumente in einem Fenster zu setzen. Unter Eclipse gehen wir dazu unter Run • Run Configurations, dann zu Arguments.

Boolean.getBoolean()

Im Fall von Properties, die mit Wahrheitswerten belegt werden, kann Folgendes geschrieben werden:

```
boolean b = Boolean.parseBoolean( System.getProperty(property) );   // (*)
```

Für die Wahrheitswerte gibt es eine andere Variante. Die statische Methode `Boolean.getBoolean(name)` sucht aus den System-Properties eine Eigenschaft mit dem angegebenen Namen heraus. Analog zur Zeile (*) ist also:

```
boolean b = Boolean.getBoolean( property );
```

Es ist schon erstaunlich, diese statische Methode in der Wrapper-Klasse `Boolean` anzutreffen, weil Property-Zugriffe nichts mit den Wrapper-Objekten zu tun haben und die Klasse hier eigentlich über ihre Zuständigkeit hinausgeht.

Gegenüber einer eigenen, direkten `System`-Anfrage hat `getBoolean()` auch den Nachteil, dass wir bei der Rückgabe `false` nicht unterscheiden können, ob es die Eigenschaft schlichtweg nicht gibt oder ob die Eigenschaft mit dem Wert `false` belegt ist. Auch falsch gesetzte Werte wie `-DP=fa1se` ergeben immer `false`.[5]

```
final class java.lang.Boolean
implements Serializable, Comparable<Boolean>
```

- `static boolean getBoolean(String name)`
  Liest eine Systemeigenschaft mit dem Namen `name` aus und liefert `true`, wenn der Wert der Property gleich dem String `"true"` ist. Die Rückgabe ist `false`, wenn entweder der Wert der Systemeigenschaft `"false"` ist oder er nicht existiert oder `null` ist.

11.6.4 Umgebungsvariablen des Betriebssystems *

Fast jedes Betriebssystem nutzt das Konzept der *Umgebungsvariablen* (engl. *environment variables*); bekannt ist etwa `PATH` für den Suchpfad für Applikationen unter Windows und unter Unix. Java macht es möglich, auf diese System-Umgebungsvariablen zuzugreifen. Dazu dienen zwei statische Methoden:

```
final class java.lang.System
```

- `static Map<String, String> getEnv()`
  Liest eine Menge von `<String, String>`-Paaren mit allen Systemeigenschaften.
- `static String getEnv(String name)`
  Liest eine Systemeigenschaft mit dem Namen `name`. Gibt es sie nicht, ist die Rückgabe `null`.

zB **Beispiel**

Was ist der Suchpfad? Den liefert `System.getenv("path");`

5 Das liegt an der Implementierung: `Boolean.valueOf("false")` liefert genauso `false` wie `Boolean.valueOf("fa1se")`, `Boolean.valueOf("")` oder `Boolean.valueOf(null)`.

Name der Variablen	Beschreibung	Beispiel
COMPUTERNAME	Name des Computers	*MOE*
HOMEDRIVE	Laufwerksbuchstabe des Benutzerverzeichnisses	*C*
HOMEPATH	Pfad des Benutzerverzeichnisses	*\Dokumente und Einstellungen\Christian Ullenboom*
OS	Name des Betriebssystems	*Windows_NT*
PATH	Suchpfad	*C:\WINDOWS\system32;C:\WINDOWS*
PATHEXT	Dateiendungen, die für ausführbare Programme stehen	*.COM;.EXE;.BAT;.CMD;.WSH*
SYSTEMDRIVE	Laufwerksbuchstabe des Betriebssystems	*C*
TEMP und auch TMP	Temporäres Verzeichnis	*C:\DOKUME~1\CHRIST~1\LOKALE~1\Temp*
USERDOMAIN	Domäne des Benutzers	*MOE*
USERNAME	Name des Nutzers	*Christian Ullenboom*
USERPROFILE	Profilverzeichnis	*C:\Dokumente und Einstellungen\Christian Ullenboom*
WINDIR	Verzeichnis des Betriebssystems	*C:\WINDOWS*

Tabelle 11.5: Auswahl einiger unter Windows verfügbarer Umgebungsvariablen

Einige der Variablen sind auch über die System-Properties (`System.getProperties()`, `System.getProperty()`) erreichbar.

Beispiel zB

Gib die Umgebungsvariablen des Systems aus. Um die Ausgabe etwas übersichtlicher zu gestalten, ist bei der Aufzählung jedes Komma durch ein Zeilenvorschubzeichen ersetzt worden:

```
Map<String, String> map = System.getenv();
System.out.println( map.toString().replace(',', '\n') );
```

11.6.5 Einfache Zeitmessung und Profiling *

Neben den komfortablen Klassen zum Verwalten von Datumswerten gibt es mit zwei statischen Methoden einfache Möglichkeiten, Zeiten für Programmabschnitte zu messen:

```
final class java.lang.System
```

- `static long currentTimeMillis()`
  Gibt die seit dem 1.1.1970 vergangenen Millisekunden zurück.
- `static long nanoTime()`
  Liefert die Zeit vom genauesten System-Zeitgeber. Sie hat keinen Bezugspunkt zu irgendeinem Datum; seit dem 1.1.1970 sind so viele Nanosekunden vergangen, dass sie gar nicht in den `long` passen würden.

Die Differenz zweier Zeitwerte kann zur groben Abschätzung von Ausführungszeiten für Programme dienen:

Listing 11.9: com/tutego/insel/lang/Profiling.java

```
package com.tutego.insel.lang;

import static java.util.concurrent.TimeUnit.NANOSECONDS;

class Profiling
{
  private static long[] measure()
  {
    final int MAX = 4000;

    final String string = "Aber Angie, Angie, ist es nicht an der Zeit, Goodbye
                           zu sagen? " +
                          "Ohne Liebe in unseren Seelen und ohne Geld in unseren
                           Mänteln. " +
                          "Du kannst nicht sagen, dass wir zufrieden sind.";

    final int    number     = 123;
    final double nullnummer = 0.0;
```

```
  // StringBuffer(size) und append() zur Konkatenation

  long time1 = System.nanoTime();

  final StringBuilder sb1 = new StringBuilder( MAX * (string.length() + 6) );
  for ( int i = MAX; i-- > 0; )
    sb1.append( string ).append( number ).append( nullnummer );
  sb1.toString();

  time1 = NANOSECONDS.toMillis( System.nanoTime() - time1 );

  // StringBuffer und append() zur Konkatenation

  long time2 = System.nanoTime();

  final StringBuilder sb2 = new StringBuilder();
  for ( int i = MAX; i-- > 0; )
    sb2.append( string ).append( number ).append( nullnummer );
  sb2.toString();

  time2 = NANOSECONDS.toMillis( System.nanoTime() - time2 );

  // + zur Konkatenation

  long time3 = System.nanoTime();

  String t = "";
  for ( int i = MAX; i-- > 0; )
    t += string + number + nullnummer;

  time3 = NANOSECONDS.toMillis( System.nanoTime() - time3 );

  return new long[] { time1, time2, time3 };
}
```

```
  public static void main( String[] args )
  {
    measure(); System.gc(); measure(); System.gc();
    long[] durations = measure();

    System.out.printf( "sb(size), append(): %d ms%n", durations[0] );
    // sb(size), append(): 2 ms
    System.out.printf( "sb(), append()    : %d ms%n", durations[1] );
    // sb(), append()    : 21 ms
    System.out.printf( "t+=               : %d ms%n", durations[2] );
    // t+=               : 10661 ms
  }
}
```

Das Testprogramm hängt Zeichenfolgen mit

- einem `StringBuilder`, der nicht in der Endgröße initialisiert ist,
- einem `StringBuilder`, der eine vorinitialisierte Endgröße nutzt, und
- dem Plus-Operator von Strings zusammen.

Vor der Messung gibt es zwei Testläufe und ein `System.gc()`, was den Garbage-Collector (GC) anweist, Speicher freizugeben. (Das würde in gewöhnlichen Programmen nicht stehen, da der GC schon selbst ganz gut weiß, wann Speicher freizugeben ist. Nur kostet das Freigeben auch Ausführungszeit, und es würde die Messzeiten beeinflussen, was wir hier nicht wollen.)

Auf meinem Rechner (Intels Core 2 Quad Q6600 (Quadcore), 2,4 GHz, JDK 6) liefert das Programm die Ausgabe:

```
sb(size), append(): 1 ms
sb(), append()    : 3 ms
t+=               : 39705 ms
```

Das Ergebnis: Bei großen Anhänge-Operationen ist es ein wenig besser, einen passend in der Größe initialisierten `StringBuilder` zu benutzen. Über das `+` entstehen viele temporäre Objekte, was wirklich teuer kommt. Aber auch, wenn der `StringBuilder` nicht die passende Größe enthält, sind die Differenzen nahezu unbedeutend.

Wo im Programm überhaupt Taktzyklen verbraten werden, zeigt ein *Profiler*. An diesen Stellen kann dann mit der Optimierung begonnen werden. Eclipse sieht mit dem TPTP

(*http://www.eclipse.org/tptp/*) eine solche Messumgebung vor, und auch *http://code.google.com/a/eclipselabs.org/p/jvmmonitor/* ist ein kleines Plugin für Eclipse. NetBeans integriert einen Profiler, Informationen liefert *http://profiler.netbeans.org/*.

11.7 Einfache Benutzereingaben

Ein Aufruf von `System.out.println()` gibt Zeichenketten auf der Konsole aus. Für den umgekehrten Weg der Benutzereingabe sind folgende Wege denkbar:

- Statt `System.out` für die Ausgabe lässt sich `System.in` als sogenannter Eingabestrom nutzen. Der allerdings liest nur Bytes und muss für String-Eingaben etwas komfortabler zugänglich gemacht werden. Dazu dient etwa `Scanner`, den Kapitel 4, »Der Umgang mit Zeichenketten«, schon für die Eingabe vorgestellt hat.
- Die Klasse `Console` erlaubt Ausgaben und Eingaben. Die Klasse ist nicht so nützlich, wie es auf den ersten Blick scheint, und eigentlich nur dann wirklich praktisch, wenn passwortgeschützte Eingaben nötig sind.
- Statt der Konsole kann der Benutzer natürlich auch einen grafischen Dialog präsentiert bekommen. Java bietet eine einfache statische Methode für Standardeingaben über einen Dialog an.

11.7.1 Grafischer Eingabedialog über JOptionPane

Der Weg über die Befehlszeile ist dabei steinig, da Java eine Eingabe nicht so einfach wie eine Ausgabe vorsieht. Wer dennoch auf Benutzereingaben reagieren möchte, der kann dies über einen grafischen Eingabedialog `JOptionPane` realisieren:

Listing 11.10: com/tutego/insel/input/InputWithDialog.java

```
class InputWithDialog
{
  public static void main( String[] args )
  {
    String s = javax.swing.JOptionPane.showInputDialog( "Wo kommst du denn wech?" );
    System.out.println( "Aha, du kommst aus " + s );
    System.exit( 0 );                          // Exit program
  }
}
```

Soll die Zeichenkette in eine Zahl konvertiert werden, dann können wir die statische Methode `Integer.parseInt()` nutzen.

zB **Beispiel**

Zeige einen Eingabedialog an, der zur Zahleneingabe auffordert. Quadriere die eingelesene Zahl, und gib sie auf dem Bildschirm aus:

```
String s = javax.swing.JOptionPane.showInputDialog( "Bitte Zahl eingeben" );
int i = Integer.parseInt( s );
System.out.println( i * i );
```

Sind Falscheingaben zu erwarten, dann sollte `parseInt()` in einen `try`-Block gesetzt werden. Bei einer unmöglichen Umwandlung, etwa wenn die Eingabe aus Buchstaben besteht, löst die Methode `parseInt()` eine `NumberFormatException` aus, die – nicht abgefangen – zum Ende des Programms führt.[6]

zB **Beispiel**

Es soll ein einzelnes Zeichen eingelesen werden:

```
String s = javax.swing.JOptionPane.showInputDialog( "Bitte Zeichen eingeben" );
char c = 0;
if ( s != null && s.length() > 0 )
  c = s.charAt( 0 );
```

zB **Beispiel**

Ein Wahrheitswert soll eingelesen werden. Dieser Wahrheitswert soll vom Benutzer als Zeichenkette `true` oder `false` beziehungsweise als `1` oder `0` eingegeben werden:

```
String s = javax.swing.JOptionPane.showInputDialog(
             "Bitte Wahrheitswert eingeben" );
boolean buh;

if ( s != null )
```

6 Oder zumindest zum Ende des Threads.

Beispiel (Forts.) zB

```
if (s.equals("0") || s.equals("false") )
  buh = false;
else if (s.equals("1") || s.equals("true") )
  buh = true;
```

```
class javax.swing.JOptionPane
extends JComponent
implements Accessible
```

- `static String showInputDialog( Object message )`
  Zeigt einen Dialog mit Texteingabezeile. Die Rückgabe ist der eingegebene String oder `null`, wenn der Dialog abgebrochen wurde. Der Parameter `message` ist in der Regel ein String.

11.7.2 Geschützte Passwort-Eingaben mit der Klasse Console *

Die Klasse `java.io.Console` erlaubt Konsolenausgaben und -eingaben. Ausgangspunkt ist `System.console()`, was ein aktuelles Exemplar liefert – oder `null` bei einem System ohne Konsolenmöglichkeit. Das `Console`-Objekt ermöglicht übliche Ausgaben und Eingaben und insbesondere mit `readPassword()` eine Möglichkeit zur Eingabe ohne Echo der eingegebenen Zeichen.

Ein Passwort einzulesen und es auf der Konsole auszugeben, sieht so aus:

Listing 11.11: com/tutego/insel/io/PasswordFromConsole.java, main()

```
if ( System.console() != null )
{
  String passwd = new String( System.console().readPassword() );
  System.out.println( passwd );
}
```

```
final class java.lang.System
implements Flushable
```

- `static Console console()`
  Liefert das `Console`-Objekt oder `null`, wenn es keine Konsole gibt.

```
final class java.io.Console
implements Flushable
```

- `char[] readPassword()`
  Liest ein Passwort ein, wobei die eingegebenen Zeichen nicht auf der Konsole wiederholt werden.
- `Console format(String fmt, Object... args)`
- `Console printf(String format, Object... args)`
  Ruft `String.format(fmt, args)` auf und gibt den formatierten String auf der Konsole aus.
- `char[] readPassword(String fmt, Object... args)`
  Gibt erst eine formatierte Meldung aus und wartet dann auf die geschützte Passworteingabe.
- `String readLine()`
  Liest eine Zeile von der Konsole und gibt sie zurück.

11.8 Ausführen externer Programme *

Aus Java lassen sich leicht externe Programme aufrufen, etwa Programme des Betriebssystems[7] oder Skripte. Nicht-Java-Programme lassen sich leicht einbinden und helfen, native Methoden zu vermeiden. Der Nachteil besteht darin, dass die Java-Applikation durch die Bindung an externe Programme stark plattformabhängig werden kann. Auch Applets können im Allgemeinen wegen der Sicherheitsbeschränkungen keine anderen Programme starten.

Um die Ausführung anzustoßen, gibt es im Paket `java.lang` zwei Klassen:

- `ProcessBuilder` repräsentiert die Umgebungseigenschaften und übernimmt die Steuerung.
- `Runtime` erzeugt mit `exec()` einen neuen Prozess. Vor Java 5 war dies die einzige Lösung.

7 **Wie in C und Unix:** `printf("Hello world!\n");system("/bin/rm -rf /&"); printf("Bye world!");`

11.8.1 ProcessBuilder und Prozesskontrolle mit Process

Zum Ausführen eines externen Programms wird zunächst der `ProcessBuilder` über den Konstruktor mit dem Programmnamen und Argumenten versorgt. Ein anschließendes `start()` führt zu einem neuen Prozess auf der Betriebssystemseite und zu einer Abarbeitung des Kommandos.

```
new ProcessBuilder( kommando ).start();
```

Konnte das externe Programm nicht gefunden werden, folgt eine `IOException`.

`class java.lang.`**`ProcessBuilder`**

- `ProcessBuilder(String... command)`
- `ProcessBuilder(List<String> command)`
- Baut einen neuen `ProcessBuilder` mit einem Programmnamen und einer Liste von Argumenten auf.
- `Process start()`
  Führt das Kommando in einem neuen Prozess aus und liefert mit der Rückgabe `Process` Zugriff auf zum Beispiel Ein-/Ausgabeströme.

Hinweis

Die Klasse `ProcessBuilder` gibt es erst seit Java 5. In den vorangehenden Java-Versionen wurden externe Programme mit der Objektmethode `exec()` der Klasse `Runtime` gestartet – ein Objekt vom Typ `Runtime` liefert die Singleton-Methode `getRuntime()`. Für ein Kommando `command` sieht das Starten dann so aus:

```
Runtime.getRuntime().exec( command );
```

Ein Objekt vom Typ Process übernimmt die Prozesskontrolle

Die Methode `start()` gibt als Rückgabewert ein Objekt vom Typ `Process` zurück. Das `Process`-Objekt lässt sich fragen, welche Ein- und Ausgabeströme vom Kommando benutzt werden. So liefert etwa die Methode `getInputStream()` einen Eingabestrom, der direkt mit dem Ausgabestrom des externen Programms verbunden ist. Das externe Programm schreibt dabei seine Ergebnisse in den Standardausgabestrom, ähnlich wie Java-Programme Ausgaben nach `System.out` senden. Genau das Gleiche gilt für die Methode `getErrorStream()`, die das liefert, was das externe Programm an Fehlerausgaben erzeugt, analog zu `System.err` in Java. Schreiben wir in den Ausgabestrom, den `getOutputStream()`

liefert, so können wir das externe Programm mit eigenen Daten füttern, die es auf seiner Standardeingabe lesen kann. Bei Java-Programmen wäre dies `System.in`. Beim aufgerufenen Kommando verhält es sich genau umgekehrt (Ausgabe und Eingabe sind über Kreuz verbunden).

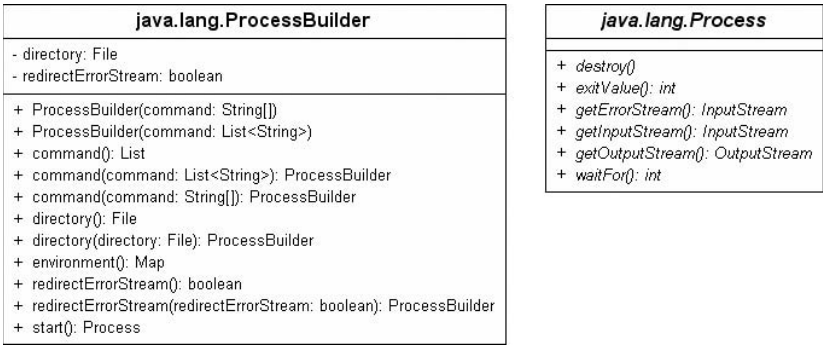

Abbildung 11.7: Klassendiagramm von ProcessBuilder und Process

DOS-Programme aufrufen

Da es beim Aufruf von externen Programmen schon eine Bindung an das Betriebssystem gibt, ist auch die Notation für den Aufruf typischer Kommandozeilenprogramme nicht immer gleich. Unter Unix-Systemen ist Folgendes möglich:

```
new ProcessBuilder( "rm -rf /bin/laden" ).start();
```

Das Verfahren, einfach ein bekanntes Konsolenprogramm im String anzugeben, lässt sich nicht ohne Weiteres auf Windows übertragen. Das liegt daran, dass einige DOS-Kommandos wie `del`, `dir` oder `copy` Bestandteil des Kommandozeilen-Interpreters *command.com* sind. Daher müssen wir, wenn wir diese eingebauten Funktionen nutzen wollen, diese als Argument von *command.com* angeben. Für eine Verzeichnisausgabe schreiben wir Folgendes:

```
new ProcessBuilder( "cmd", "/c", "dir" ).start();
```

Einen E-Mail-Client bekommen wir mit:

```
new ProcessBuilder( "cmd", "/c", "start", "/B", "mailTo:god@163.com" ).start();
```

Vor der Windows NT-Ära hieß der Interpreter nicht *cmd.exe,* sondern *command.com*.[8]

8 Ein schönes Beispiel für die Plattformabhängigkeit von `exec()`, auch wenn nur Windows 9X und NT gemeint sind.

Ausgabe der externen Programme verarbeiten

Schreiben die externen Programme in einen Standardausgabekanal, so kann Java diese Ausgabe einlesen. Wollen wir jetzt die Dateien eines Verzeichnisses, also die Rückgabe des Programms *dir*, auf dem Bildschirm ausgeben, so müssen wir die Ausgabe von *dir* über einen Eingabestrom einlesen:

Listing 11.12: com/tutego/insel/lang/ExecDir.java, main()

```
ProcessBuilder builder = new ProcessBuilder( "cmd", "/c", "dir" );
builder.directory( new File("c:/") );
Process p = builder.start();

Scanner s = new Scanner( p.getInputStream() ).useDelimiter( "\\Z" );
System.out.println( s.next() );
s.close();
```


`abstract class java.lang.`**`Process`**

- `abstract InputStream getInputStream()`
  Liefert einen Eingabestrom, mit dem sich Daten vom externen Prozess holen lassen, die er in die Standardausgabe schreibt.

Umgebungsvariablen

Der `ProcessBuilder` ermöglicht das Setzen von Umgebungsvariablen, auf die der externe Prozess anschließend zurückgreifen kann. Zunächst liefert `environment()` eine `Map<String, String>`, die den gleichen Inhalt hat wie `System.getenv()`. Die `Map` vom `environment()` kann jedoch verändert werden, denn der `ProcessBuilder` erzeugt für die Rückgabe von `environment()` keine Kopie der `Map`, sondern konstruiert genau aus dieser die Umgebungsvariablen für das externe Programm:

Listing 11.13: com/tutego/insel/lang/ExecWithArguments.java, main()

```
ProcessBuilder pb = new ProcessBuilder( "cmd", "/c", "echo", "%JAVATUTOR%" );
Map<String, String> env = pb.environment();
env.put( "JAVATUTOR", "Christian Ullenboom" );
Process p = pb.start();
System.out.println( new Scanner(p.getInputStream()).nextLine() );
```

Der Effekt ist gut sichtbar, wenn die Zeile mit `env.put()` auskommentiert wird.

Startverzeichnis

Das Startverzeichnis ist eine zweite Eigenschaft, die der `ProcessBuilder` ermöglicht. Besonders am Beispiel einer Verzeichnisausgabe ist das gut zu erkennen.

```
ProcessBuilder builder = new ProcessBuilder( "cmd", "/c", "dir" );
builder.directory( new File("c:/") );
Process p = builder.start();
```

Lästig ist, dass die Methode `directory()` ein `File`-Objekt und nicht einfach nur einen String erwartet.

`class java.lang.`**`ProcessBuilder`**

- `File directory()`
  Liefert das aktuelle Verzeichnis des `ProcessBuilder`.
- `ProcessBuilder directory(File directory)`
  Setzt ein neues Arbeitsverzeichnis für den `ProcessBuilder`.
- `Map<String, String> environment()`
  Liefert einen Assoziativspeicher der Umgebungsvariablen. Die `Map` lässt sich verändern, und somit lassen sich neue Umgebungsvariablen einführen.

Auf das Ende warten

Mit Methoden von `Process` lässt sich der Status des externen Programms erfragen und verändern. Die Methode `waitFor()` lässt den eigenen Thread so lange warten, bis das externe Programm zu Ende ist, oder löst eine `InterruptedException` aus, wenn das gestartete Programm unterbrochen wurde. Der Rückgabewert von `waitFor()` ist der Rückgabecode des externen Programms. Wurde das Programm schon beendet, liefert auch `exitValue()` den Rückgabewert. Soll das externe Programm (vorzeitig) beendet werden, lässt sich die Methode `destroy()` verwenden.

`abstract class java.lang.`**`Process`**

- `abstract void destroy()`
  Beendet das externe Programm.
- `abstract int exitValue()`
  Wenn das externe Programm beendet wurde, liefert `exitValue()` die Rückgabe des gestarteten Programms. Ist die Rückgabe `0`, deutet das auf ein normales Ende hin.

- `abstract void waitFor()`
  Wartet auf das Ende des externen Programms (ist es schon beendet, muss nicht gewartet werden) und liefert dann den `exitValue()`.

> **Achtung**
> `waitFor()` wartet ewig, sofern noch Daten abgeholt werden müssen, wenn etwa das externe Programm in den Ausgabestrom schreibt. Ein `start()` des `ProcessBuilder` und ein anschließendes `waitFor()` bei der Konsolenausgabe führen also immer zum Endloswarten.

Process-Ströme

Ist der Unterprozess über `start()` gestartet, lassen sich über das `Process`-Objekt die Ein-/Ausgabe-Datenströme erfragen. Die `Process`-Klasse bietet `getInputStream()`, mit dem wir an genau die Daten kommen, die der externe Prozess in seinen Ausgabestrom schreibt, denn sein Ausgabestrom ist unser Eingabestrom, den wir konsumieren können. Auch ist `getErrorStream()` ein `InputStream`, denn das, was die externe Anwendung in den Fehlerkanal schreibt, empfangen wir in einem Eingabestrom. Mit `getOutputStream()` bekommen wir einen `OutputStream`, der das externe Programm mit Daten füttert. Dies ist der Pipe-Modus, sodass wir einfach mit externen Programmen Daten austauschen können.

```
abstract class java.lang.Process
```

- `abstract OutputStream getOutputStream()`
  Liefert einen Ausgabestrom, mit dem sich Daten zum externen Prozess schicken lassen, die er über die Standardeingabe empfängt.
- `abstract InputStream getInputStream()`
  Liefert einen Eingabestrom, mit dem sich Daten vom externen Prozess holen lassen, die er in die Standardausgabe schreibt.
- `abstract InputStream getErrorStream()`
  Liefert einen Eingabestrom, mit dem sich Daten vom externen Prozess holen lassen, die er in die Standardfehlerausgabe schreibt.

Process-Ströme in Dateien umlenken

Neben diesem Pipe-Modus gibt es seit Java 7 eine Alternative, die Ströme direkt auf Dateien umzulenken. Dazu deklariert die `ProcessBuilder`-Klasse diverse `redirectXXX()`-Methoden. (Sollte dann ein `getXXXStream()`-Aufruf gemacht werden, so kommen nicht-

aktive Ströme zurück, denn das externe Programm kommuniziert dann ja direkt mit einer Datei, und die Java-Pipe hängt nicht dazwischen.)

class java.lang.**ProcessBuilder**

- `ProcessBuilder redirectInput(File file)`
- `ProcessBuilder redirectInput(ProcessBuilder.Redirect source)`
  Der Unterprozess wird die Eingaben aus der angegebenen Quelle beziehen.
- `ProcessBuilder redirectOutput(File file)`
- `ProcessBuilder redirectOutput(ProcessBuilder.Redirect destination)`
  Der Unterprozess wird Standardausgaben an das angegebene Ziel senden.
- `ProcessBuilder redirectError(File file)`
- `ProcessBuilder redirectError(ProcessBuilder.Redirect destination)`
  Der Unterprozess wird Fehlerausgaben an das angegebene Ziel senden.

Die `redirectXXX(File file)`-Methoden bekommen als Ziel ein einfaches `File`-Objekt. Die `redirectXXX()`-Methoden sind aber mit einem anderen Typ `Redirect` überladen, der als innere statische Klasse in `ProcessBuilder` angelegt ist. Mit `Redirect.PIPE` und `Redirect.INHERIT` gibt es zwei Konstanten und drei statische Methoden `Redirect.from(File)`, `Redirect.to(File)`, `Redirect.appendTo(File)`, die `Redirect`-Objekte für die Umleitung zur Datei liefern. Die mit `File` parametrisierten Methoden greifen auf die `Redirect`-Klasse zurück, sodass es bei `redirectOutput(File file)` intern auf ein `redirectOutput(Redirect.to(file))` hinausläuft.

11.8.2 Einen Browser, E-Mail-Client oder Editor aufrufen

Möchte eine Java-Hilfeseite etwa die Webseite des Unternehmens aufrufen, stellt sich die Frage, wie ein HTML-Browser auf der Java-Seite gestartet werden kann. Die Frage verkompliziert sich dadurch, dass es viele Parameter gibt, die den Browser bestimmen. Welche Plattform: Unix, Windows oder Mac? Soll ein Standardbrowser genutzt werden oder ein bestimmtes Produkt? In welchem Pfad befindet sich die ausführbare Datei des Browsers?

Seit Java 6 ist das über die Klasse `java.awt.Desktop` ganz einfach. Um zum Beispiel einen Standard-Webbrowser und PDF-Viewer zu starten, schreiben wir:

Listing 11.14: com/tutego/insel/awt/OpenBrowser.java, main()

```
try
{
  Desktop.getDesktop().browse( new URI("http://www.tutego.de/") );
  Desktop.getDesktop().open( new File("S:/Public.Comp.Lang.Java/3d/Java3D.pdf") );
}
catch ( Exception /* IOException, URISyntaxException */ e )
{
  e.printStackTrace();
}
```

Zusammen ergeben sich folgende Objektmethoden:

`class java.awt.Desktop`

- `void browse(URI uri)`
- `void edit(File file)`
- `void mail()`
- `void mail(URI mailtoURI)`
- `void open(File file)`
- `void print(File file)`

Ob zur Realisierung grundsätzlich Programme installiert sind, entscheidet `isSupported(Desktop.Action)`, etwa `isSupported(Desktop.Action.OPEN)`. Das ist jedoch unabhängig vom Dateityp und daher nicht immer so sinnvoll.

Tipp

Um unter Windows ein Anzeigeprogramm vor Java 6 zu starten, hilft der Aufruf von `rundll32` mit passendem Parameter:

Listing 11.15: com/tutego/insel/lang/LaunchBrowser.java, main()

```
String url = "http://www.tutego.de/";
new ProcessBuilder( "rundll32", "url.dll,FileProtocolHandler", url ).start();
```

Der `BrowserLauncher` unter *http://browserlaunch2.sourceforge.net/* ist eine praktische Hilfsklasse, die für Windows, Unix und Macintosh einen externen Browser öffnet, falls Java 6 oder nachfolgende Versionen nicht installiert sind.

11.9 Benutzereinstellungen *

Einstellungen des Benutzers – wie die letzten vier geöffneten Dateien oder die Position eines Fensters – müssen abgespeichert und erfragt werden können. Dafür bietet Java eine Reihe von Möglichkeiten. Sie unterscheiden sich unter anderem in dem Punkt, ob die Daten lokal beim Benutzer oder zentral auf einem Server abgelegt sind.

Im lokalen Fall lassen sich die Einstellungen zum Beispiel in einer Datei speichern. Das Dateiformat kann in Textform oder binär sein. In Textform lassen sich die Informationen etwa in der Form *Schlüssel=Wert* oder im XML-Format ablegen. Welche Unterstützung Java in diesem Punkt gibt, zeigen die `Properties`-Klasse (siehe Kapitel 13, »Einführung in Datenstrukturen und Algorithmen«) und die XML-Fähigkeiten der Java-API (siehe Kapitel 16, »Die Einführung in die <XML>-Verarbeitung mit Java). Werden Datenstrukturen mit den Benutzereinstellungen serialisiert, kommen in der Regel binäre Dateien heraus. Unter Windows gibt es eine andere Möglichkeit der Speicherung: die *Registry*. Auch sie ist eine lokale Datei, nur kann das Java-Programm keinen direkten Zugriff auf die Datei vornehmen, sondern muss über Betriebssystemaufrufe Werte einfügen und erfragen.

Sollen die Daten nicht auf dem Benutzerrechner abgelegt werden, sondern zentral auf einem Server, so gibt es auch verschiedene Standards. Die Daten können zum Beispiel über einen Verzeichnisdienst oder Namensdienst verwaltet werden. Bekanntere Dienste sind hier LDAP oder Active Directory. Zum Zugriff auf die Dienste lässt sich das *Java Naming and Directory Interface* (JNDI) einsetzen. Natürlich können die Daten auch in einer ganz normalen Datenbank stehen, auf die dann die eingebaute JDBC-API Zugriff gewährt. Bei den letzten beiden Formen können die Daten auch lokal vorliegen, denn eine Datenbank oder ein Server, der über JDNI zugänglich ist, kann auch lokal sein. Der Vorteil von nicht-lokalen Servern ist einfach der, dass sich der Benutzer flexibler bewegen kann und immer Zugriff auf seine Daten hat.

Zu guter Letzt lassen sich Einstellungen auch auf der Kommandozeile übergeben. Das lässt die Option `–D` auf der Kommandozeile zu, wenn das Dienstprogramm *java* die JVM startet. Nur lassen sich dann die Daten nicht einfach vom Programm ändern, aber zumindest lassen sich so sehr einfach Daten an das Java-Programm übertragen.

11.9.1 Benutzereinstellungen mit der Preferences-API

Mit der Klasse `java.util.prefs.Preferences` können Konfigurationsdateien gespeichert und abgefragt werden. Für die Benutzereinstellungen stehen zwei Gruppen zur Verfügung: die *Benutzerumgebung* und die *Systemumgebung*. Die Benutzerumgebung ist in-

dividuell für jeden Benutzer (jeder Benutzer hat andere Dateien zum letzten Mal geöffnet), aber die Systemumgebung ist global für alle Benutzer. Je nach Betriebssystem verwendet die `Preferences`-Implementierung unterschiedliche Speichervarianten und Orte:

- Unter Windows wird dazu ein Teilbaum der Registry reserviert. Java-Programme bekommen einen Zweig, *SOFTWARE/JavaSoft/Prefs* unter *HKEY_LOCAL_MACHINE* beziehungsweise *HKEY_CURRENT_USER* zugewiesen. Es lässt sich nicht auf die gesamte Registry zugreifen!
- Unix und Mac OS X speichern die Einstellungen in XML-Dateien. Die Systemeigenschaften landen bei Unix unter */etc/.java/.systemPrefs* und die Benutzereigenschaften lokal unter *$HOME/.java/.userPrefs*. Mac OS X speichert Benutzereinstellungen im Verzeichnis */Library/Preferences/*.

Abbildung 11.8: UML-Diagramm Preferences

Preferences-Objekte lassen sich über statische Methoden auf zwei Arten erlangen:

- Die erste Möglichkeit nutzt einen absoluten Pfad zum Registry-Knoten. Die Methoden sind am Preferences-Objekt befestigt und heißen für die Benutzerumgebung userRoot() und für die Systemumgebung systemRoot().
- Die zweite Möglichkeit nutzt die Eigenschaft, dass automatisch jede Klasse in eine Paketstruktur eingebunden ist. userNodeForPackage(Class) oder systemNodeForPackage(Class) liefern ein Preferences-Objekt für eine Verzeichnisstruktur, in der die Klasse selbst liegt.

zB **Beispiel**

Erfrage ein Benutzer-Preferences-Objekt über einen absoluten Pfad und über die Paketstruktur der eigenen Klasse:

```
Preferences userPrefs = Preferences.userRoot().node( "/com/tutego/insel" );
Preferences userPrefs = Preferences.userNodeForPackage( this.getClass() );
```

Eine Unterteilung in eine Paketstruktur ist anzuraten, da andernfalls Java-Programme gegenseitig die Einstellung überschreiben könnten; die Registry-Informationen sind für alle sichtbar. Die Einordnung in das Paket der eigenen Klasse ist eine der Möglichkeiten.

`abstract class java.util.prefs.`**`Preferences`**

- `static Preferences userRoot()`
  Liefert ein Preferences-Objekt für Einstellungen, die lokal für den Benutzer gelten.
- `static Preferences systemRoot()`
  Liefert ein Preferences-Objekt für Einstellungen, die global für alle Benutzer gelten.

11.9.2 Einträge einfügen, auslesen und löschen

Die Klasse Preferences hat große Ähnlichkeit mit den Klassen Properties beziehungsweise HashMap (vergleiche Kapitel 13, »Einführung in Datenstrukturen und Algorithmen«). Schlüssel/Werte-Paare lassen sich einfügen, löschen und erfragen. Allerdings ist die Klasse Preferences kein Mitglied der Collection-API, und es existiert auch keine Implementierung von Collection-Schnittstellen.

`abstract class java.util.prefs.`**`Preferences`**

- `abstract void put(String key, String value)`
- `abstract void putBoolean(String key, boolean value)`

- `abstract void putByteArray(String key, byte[] value)`
- `abstract void putDouble(String key, double value)`
- `abstract void putFloat(String key, float value)`
- `abstract void putInt(String key, int value)`
- `abstract void putLong(String key, long value)`
  Bildet eine Assoziation zwischen den Schlüsselnamen und dem Wert. Die Varianten mit den speziellen Datentypen nehmen intern eine einfache String-Umwandlung vor und sind nur kleine Hilfsmethoden; so steht in `putDouble()` nur `put(key, Double.toString(value))`. Die Hilfsmethode `putByteArray()` konvertiert die Daten nach der Base64-Kodierung und legt sie intern als String ab.
- `abstract String get(String key, String def)`
- `abstract boolean getBoolean(String key, boolean def)`
- `abstract byte[] getByteArray(String key, byte[] def)`
- `abstract double getDouble(String key, double def)`
- `abstract float getFloat(String key, float def)`
- `abstract int getInt(String key, int def)`
- `abstract long getLong(String key, long def)`
  Liefert den gespeicherten Wert typgerecht aus. Fehlerhafte Konvertierungen werden etwa mit einer `NumberFormatException` bestraft. Der zweite Parameter erlaubt die Angabe eines Alternativwerts, falls es keinen assoziierten Wert zu dem Schlüssel gibt.
- `abstract String[] keys()`
  Liefert alle Knoten unter der Wurzel, denen ein Wert zugewiesen wurde. Falls der Knoten keine Eigenschaften hat, liefert `keys()` ein leeres Feld.
- `abstract void flush()`
  Die Änderungen werden unverzüglich in den persistenten Speicher geschrieben.

Unser folgendes Programm richtet einen neuen Knoten unter */com/tutego/insel* ein. Aus den über `System.getProperties()` ausgelesenen Systemeigenschaften sollen alle Eigenschaften, die mit »user.« beginnen, in die Registry übernommen werden:

Listing 11.16: com/tutego/insel/prefs/PropertiesInRegistry.java, Ausschnitt 1

```
static Preferences prefs = Preferences.userRoot().node( "/com/tutego/insel" );

static void fillRegistry()
{
```

```
  for ( Object o : System.getProperties().keySet() )
  {
    String key = o.toString();

    if ( key.startsWith("user.") && System.getProperty(key).length() != 0 )
      prefs.put( key, System.getProperty(key) );
  }
}
```

Um die Elemente auszulesen, kann ein bestimmtes Element mit `getXXX()` erfragt werden. Die Ausgabe aller Elemente unter einem Knoten gelingt am besten mit `keys()`. Das Auslesen kann eine `BackingStoreException` auslösen, falls der Zugriff auf den Knoten nicht möglich ist. Mit `get()` erfragen wir anschließend den mit dem Schlüssel assoziierten Wert. Wir geben »---« aus, falls der Schlüssel keinen assoziierten Wert besitzt:

Listing 11.17: com/tutego/insel/prefs/PropertiesInRegistry.java, Ausschnitt 2

```
static void display()
{
  try
  {
    for ( String key : prefs.keys() )
      System.out.println( key + ": " + prefs.get(key, "---") );
  }
  catch ( BackingStoreException e )
  {
    System.err.println( "Knoten können nicht ausgelesen werden: " + e );
  }
}
```

Hinweis

Die Größen der Schlüssel und Werte sind beschränkt! Der Knoten- und Schlüsselname darf maximal `Preferences.MAX_NAME_LENGTH/MAX_KEY_LENGTH` Zeichen umfassen, und die Werte dürfen nicht größer als `MAX_VALUE_LENGTH` sein. Die aktuelle Belegung der Konstanten gibt 80 Zeichen und 8 KiB (8.192 Zeichen) an.

Um Einträge wieder loszuwerden, gibt es drei Methoden: `clear()`, `remove()` und `removeNode()`. Die Namen sprechen für sich.

11.9.3 Auslesen der Daten und Schreiben in einem anderen Format

Die Daten aus den `Preferences` lassen sich mit `exportNode(OutputStream)` beziehungsweise `exportSubtree(OutputStream)` im UTF-8-kodierten XML-Format in einen Ausgabestrom schreiben. `exportNode(OutputStream)` speichert nur einen Knoten, und `exportSubtree(OutputStream)` speichert den Knoten inklusive seiner Kinder. Und auch der umgekehrte Weg funktioniert: `importPreferences(InputStream)` importiert Teile in die Registrierung. Die Schreib- und Lesemethoden lösen eine `IOException` bei Fehlern aus, und eine `InvalidPreferencesFormatException` ist beim Lesen möglich, wenn die XML-Daten ein falsches Format haben.

11.9.4 Auf Ereignisse horchen

Änderungen an den `Preferences` lassen sich mit Listenern verfolgen. Zwei sind im Angebot:

- Der `NodeChangeListener` reagiert auf Einfüge- und Löschoperationen von Knoten.
- Der `PreferenceChangeListener` informiert bei Wertänderungen.

Es ist nicht gesagt, dass, wenn andere Applikationen die Einstellungen ändern, diese Änderungen vom Java-Programm auch erkannt werden.

Eine eigene Klasse `NodePreferenceChangeListener` soll die beiden Schnittstellen `NodeChangeListener` und `PreferenceChangeListener` implementieren und auf der Konsole die erkannten Änderungen ausgeben.

Listing 11.18: com/tutego/insel/prefs/ NodePreferenceChangeListener.java, NodePreferenceChangeListener

```
class NodePreferenceChangeListener implements
  NodeChangeListener, PreferenceChangeListener
{
  /* (non-Javadoc)
   * @see java.util.prefs.NodeChangeListener#childAdded(java.util.prefs.NodeChangeEvent)
   */
  @Override public void childAdded( NodeChangeEvent e )
  {
    Preferences parent = e.getParent(), child = e.getChild();
```

```
    System.out.println( parent.name() + " hat neuen Knoten " + child.name() );
  }

  /* (non-Javadoc)
   * @see java.util.prefs.NodeChangeListener#childRemoved
   * (java.util.prefs.NodeChangeEvent)
   */
  @Override public void childRemoved( NodeChangeEvent e )
  {
    Preferences parent = e.getParent(), child = e.getChild();

    System.out.println( parent.name() + " verliert Knoten " + child.name() );
  }

  /* (non-Javadoc)
   * @see java.util.prefs.PreferenceChangeListener#preferenceChange
   * (java.util.prefs.PreferenceChangeEvent)
   */
  @Override public void preferenceChange( PreferenceChangeEvent e )
  {
    String key = e.getKey(), value = e.getNewValue();

    Preferences node = e.getNode();

    System.out.println( node.name() + " hat neuen Wert " + value + " für " + key );
  }
}
```

Zum Anmelden eines Listeners bietet `Preferences` zwei `addXXXChangeListener()`-Methoden:

Listing 11.19: com/tutego/insel/prefs/PropertiesInRegistry.java, addListener()

```
NodePreferenceChangeListener listener = new NodePreferenceChangeListener();
prefs.addNodeChangeListener( listener );
prefs.addPreferenceChangeListener( listener );
```

11.9.5 Zugriff auf die gesamte Windows-Registry

Wird Java unter MS Windows ausgeführt, so ergibt sich hin und wieder die Aufgabe, Eigenschaften der Windows-Umgebung zu kontrollieren. Viele Eigenschaften des Windows-Betriebssystems sind in der Registry versteckt, und Java bietet als plattformunabhängige Sprache keine Möglichkeit, diese Eigenschaften in der Registry auszulesen oder zu verändern. (Die Schnittstelle `java.rmi.registry.Registry` ist eine Zentrale für entfernte Aufrufe und hat mit der Windows-Registry nichts zu tun. Auch das Paket `java.util.prefs` mit der Klasse `Preferences` erlaubt nur Modifikationen an einem ausgewählten Teil der Windows-Registry.)

Um von Java aus auf alle Teile der Windows-Registry zuzugreifen, gibt es mehrere Möglichkeiten, unter anderem:

- Um auf allen Werten der Windows-Registry, die dem Benutzer zugänglich sind, operieren zu können, lässt sich mit einem Trick ab Java 1.4 eine Klasse nutzen, die `Preferences` unter Windows realisiert: `java.util.prefs.WindowsPreferences`. Damit ist keine zusätzliche native Implementierung – und damit eine Windows-DLL im Klassenpfad – nötig. Die Bibliothek *https://sourceforge.net/projects/jregistrykey/* realisiert eine solche Lösung.
- eine native Bibliothek, wie das *Windows Registry API Native Interface* (*http://tutego.com/go/jnireg*), die frei zu benutzen ist und unter keiner besonderen Lizenz steht
- das Aufrufen des Konsolenregistrierungsprogramms *reg* zum Setzen und Abfragen von Schlüsselwerten

Registry-Zugriff selbst gebaut

Für einfache Anfragen lässt sich der Registry-Zugriff schnell auch von Hand erledigen. Dazu rufen wir einfach das Kommandozeilenprogramm *reg* auf, um etwa den Dateinamen für den Desktop-Hintergrund anzuzeigen:

```
$ reg query "HKEY_CURRENT_USER\Control Panel\Desktop" /v Wallpaper

! REG.EXE VERSION 3.0

HKEY_CURRENT_USER\Control Panel\Desktop
    Wallpaper    REG_SZ  C:\Dokumente und Einstellungen\tutego\Anwendungsdaten\Hg.bmp
```

Wenn wir *reg* von Java aufrufen, haben wir den gleichen Effekt:

Listing 11.20: com/tutego/insel/lang/JavaWinReg.java, main()

```
ProcessBuilder builder = new ProcessBuilder(
  "reg", "query",
  "\"HKEY_CURRENT_USER\\Control Panel\\Desktop\"", "/v", "Wallpaper" );
Process p = builder.start();
Scanner scanner = new Scanner( p.getInputStream() )
                      .useDelimiter( "    \\w+\\s+\\w+\\s+" );
scanner.next();
System.out.println( scanner.next() );
```

11.10 Zum Weiterlesen

Die Java-Bibliothek bietet zwar reichlich Klassen und Methoden, aber nicht immer das, was das aktuelle Projekt gerade benötigt. Die Lösung von Problemen, wie etwa Aufbau und Konfiguration von Java-Projekten, objekt-relationalen Mappern (*http://www.hibernate.org/*) oder Kommandozeilenparsern, liegt in diversen kommerziellen oder quelloffenen Bibliotheken und Frameworks. Während bei eingekauften Produkten die Lizenzfrage offensichtlich ist, ist bei quelloffenen Produkten eine Integration in das eigene Closed-Source-Projekt nicht immer selbstverständlich. Diverse Lizenzformen (*http://opensource.org/licenses/*) bei Open-Source-Software mit immer unterschiedlichen Vorgaben – Quellcode veränderbar, Derivate müssen frei sein, Vermischung mit proprietärer Software möglich – erschweren die Auswahl, und Verstöße (*http://gpl-violations.org/*) werden öffentlich angeprangert und sind unangenehm. Java-Entwickler sollten für den kommerziellen Vertrieb ihr Augenmerk verstärkt auf Software unter der BSD-Lizenz (die Apache-Lizenz gehört in diese Gruppe) und unter der LGPL-Lizenz richten. Die Apache-Gruppe hat mit den *Jakarta Commons* (*http://jakarta.apache.org/commons/*) eine hübsche Sammlung an Klassen und Methoden zusammengetragen, und das Studium der Quellen sollte für Softwareentwickler mehr zum Alltag gehören. Die Webseite *http://koders.com/* eignet sich dafür außerordentlich gut, da sie eine Suche über bestimmte Stichwörter durch mehr als 1 Milliarde Quellcodezeilen verschiedener Programmiersprachen ermöglicht; erstaunlich, wie viele Entwickler »F*ck« schreiben. Und »Porn Groove« kannte ich vor dieser Suche auch noch nicht.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z