

Christian Ullenboom

Java

ist auch eine Insel

Einführung, Ausbildung, Praxis

- ▶ Programmieren mit der Java Platform, Standard Edition 25
- ▶ Java von A bis Z: Einführung, Praxis, Referenz
- ▶ Sprachgrundlagen, OOP, Datenstrukturen, Unit-Tests u. v. m.

Das
Standard-
werk

18., aktualisierte und überarbeitete Auflage



Rheinwerk
Computing

Kapitel 3

Klassen und Objekte

»Nichts auf der Welt ist so gerecht verteilt wie der Verstand. Denn jedermann ist davon überzeugt, dass er genug davon habe.«

– René Descartes (1596–1650)

Java ist eine objektorientierte Programmiersprache, und dieses Kapitel setzt Objekte in den Mittelpunkt, die durch einen Bauplan (die Klasse) erzeugt werden. Objekte werden über Referenzen angesprochen, und über Referenzen lassen sich Objekte an andere Stellen weiterreichen und vergleichen.

3.1 Objektorientierte Programmierung (OOP)

In einem Buch über Java-Programmierung müssen mehrere Teile vereinigt werden:

- ▶ zunächst die grundsätzliche Programmierung nach dem imperativen Prinzip (Variablen, Operatoren, Fallunterscheidung, Schleifen, einfache statische Methoden) in einer neuen Grammatik für Java,
- ▶ dann die Objektorientierung (Objekte, Klassen, Vererbung, Schnittstellen), erweiterte Möglichkeiten der Java-Sprache (Ausnahmen, Generics, Lambda-Ausdrücke) und zum Schluss
- ▶ die Bibliotheken (String-Verarbeitung, Ein-/Ausgabe ...).

Dieses Kapitel stellt das Paradigma der Objektorientierung in den Mittelpunkt und zeigt die Syntax, wie etwa in Java Klassen realisiert werden und Klassen-/Objektvariablen sowie Methoden eingesetzt werden.

Hinweis

Java ist natürlich nicht die erste objektorientierte Sprache (OO-Sprache), auch C++ war nicht die erste. Klassischerweise gelten Smalltalk und insbesondere Simula-67 aus dem Jahr 1967 als Stammväter aller OO-Sprachen. Die eingeführten Konzepte sind bis heute aktuell, darunter die vier allgemein anerkannten Prinzipien der OOP: *Abstraktion*, *Kapselung*, *Vererbung* und *Polymorphie*.¹



¹ Keine Sorge, alle vier Grundsäulen werden in den nächsten Kapiteln ausführlich beschrieben!

3.1.1 Warum überhaupt OOP?

Da Menschen die Welt in Objekten wahrnehmen, wird auch die Analyse von Systemen häufig schon objektorientiert modelliert. Doch mit prozeduralen Systemen, die lediglich Unterprogramme als Ausdrucksmittel haben, wird die Abbildung des objektorientierten Designs in eine Programmiersprache schwer, und es entsteht ein Bruch. Im Laufe der Zeit entwickeln sich Dokumentation und Implementierung auseinander; die Software ist dann schwer zu warten und zu erweitern. Besser ist es, objektorientiert zu denken und dann eine objektorientierte Programmiersprache zur Abbildung zu haben.



Hinweis

Bad code can be written in any language. (Zu Deutsch: Schlechter Code kann in jeder Sprache geschrieben werden.)

Identität, Zustand, Verhalten

Die in der Software abgebildeten Objekte haben drei wichtige Eigenschaften:

- ▶ Jedes Objekt hat eine Identität.
- ▶ Jedes Objekt hat einen Zustand.
- ▶ Jedes Objekt zeigt ein Verhalten.

Diese drei Eigenschaften haben wichtige Konsequenzen: zum einen, dass die Identität des Objekts während seines Lebens bis zu seinem Tod dieselbe bleibt und sich nicht ändern kann. Zum anderen werden die Daten und der Programmcode zur Manipulation dieser Daten als zusammengehörig behandelt. In prozeduralen Systemen finden sich oft Szenarien wie das folgende: Es gibt einen großen Speicherbereich, auf den alle Unterprogramme irgendwie zugreifen können. Bei den Objekten ist das anders, da sie logisch ihre eigenen Daten verwalten und die Manipulation überwachen.

In der objektorientierten Softwareentwicklung geht es also darum, in Objekten zu modellieren und dann zu programmieren. Das Design nimmt dabei eine zentrale Stellung ein; große Systeme werden zerlegt und immer feiner beschrieben. Hier passt sehr gut die Aussage des französischen Schriftstellers François Duc de La Rochefoucauld (1613–1680):

»Wer sich zu viel mit dem Kleinen abgibt, wird unfähig für Großes.«

3.1.2 Denk ich an Java, denk ich an Wiederverwendbarkeit

Bei jedem neuen Projekt fällt auf, dass in früheren Projekten schon ähnliche Probleme gelöst werden mussten. Natürlich sollen bereits gelöste Probleme nicht neu implementiert, sondern sich wiederholende Teile bestmöglich in unterschiedlichen Kontexten wiederverwendet werden; das Ziel ist die bestmögliche Wiederverwendung von Komponenten.

Wiederverwendbarkeit von Programmteilen existiert nicht erst seit der objektorientierten Programmierung, doch objektorientierte Sprachen erleichtern die Umsetzung wiederverwendbarer Softwarekomponenten erheblich. Ein Beispiel dafür sind die vielen Tausend Klassen der Java-Bibliothek: Sie stellen gebrauchsfertige Lösungen bereit, sodass grundlegende Aufgaben wie Datenstrukturen oder die Pufferung von Datenströmen nicht immer neu implementiert werden müssen.

Auch wenn Java eine objektorientierte Programmiersprache ist, ist das kein Garant für tolles Design und optimale Wiederverwendbarkeit. Eine objektorientierte Programmiersprache erleichtert objektorientiertes Programmieren, aber in einer einfachen Programmiersprache wie C lässt sich ebenfalls objektorientiert programmieren. In Java sind auch Programme möglich, die aus nur einer Klasse bestehen und dort 5.000 Zeilen Programmcode mit statischen Methoden unterbringen. Bjarne Stroustrup (der Schöpfer von C++, von seinen Freunden auch Stumpy genannt) sagte treffend über den Vergleich von C und C++:

»C makes it easy to shoot yourself in the foot, C++ makes it harder, but when you do, it blows away your whole leg.«²

Im Sinne unserer didaktischen Vorgehensweise wird dieses Kapitel zunächst einige Klassen der Standardbibliothek verwenden. Wir beginnen mit der Klasse `Point`, die zweidimensionale Punkte repräsentiert. In einem zweiten Schritt werden wir eigene Klassen programmieren. Anschließend kümmern wir uns um das Konzept der Abstraktion in Java, nämlich darum, wie Gruppen zusammenhängender Klassen gestaltet werden.

3.2 Eigenschaften einer Klasse

Klassen sind ein wichtiges Merkmal objektorientierter Programmiersprachen. Eine Klasse definiert einen neuen Typ, beschreibt die Eigenschaften der Objekte und gibt somit den Bauplan an.

Jedes Objekt ist ein *Exemplar* (auch *Instanz*³ oder *Ausprägung* genannt) einer Klasse.

Eine Klasse deklariert im Wesentlichen zwei Dinge:

- ▶ Attribute (was das Objekt hat)
- ▶ Operationen (was das Objekt kann)

Attribute und Operationen heißen auch *Eigenschaften* eines Objekts; manchmal werden jedoch auch nur Attribute Eigenschaften genannt. Welche Eigenschaften eine Klasse tatsäch-

² Oder wie es Bertrand Meyer sagt: *»Do not replace legacy software by lega-c++ software.«*

³ Ich vermeide das Wort *Instanz* und verwende dafür durchgängig das Wort *Exemplar*. An die Stelle von *instanziierten* tritt das einfache Wort *erzeugen*. *Instanz* ist eine irreführende Übersetzung des englischen Ausdrucks *»instance«*.

lich besitzen soll, wird in der Analyse- und Designphase festgelegt. Diese wird in diesem Buch kein Thema sein; für uns liegen die Klassenbeschreibungen schon vor.

Die Operationen einer Klasse setzt die Programmiersprache Java durch *Methoden* um. Die Attribute eines Objekts definieren die Zustände, und sie werden durch Klassen-/Objektvariablen implementiert (die auch *Felder*⁴ genannt werden).



Hinweis

Im Begriff »objektorientierte Programmierung« taucht zwar der Begriff »Objekt« auf, aber nicht der Begriff »Klasse«, den wir auch schon oft verwendet haben. Warum heißt es also nicht stattdessen »klassenbasierte Programmierung«? Der Grund ist, dass Klassendeklarationen für objektorientierte Programme nicht zwingend nötig sind. Ein anderer Ansatz ist die *prototypbasierte objektorientierte Programmierung*. Hier ist JavaScript der bekannteste Vertreter; dabei gibt es nur Objekte, und die sind mit einer Art Basistyp, dem *Prototyp*, verkettet.

Um sich einer Klasse zu nähern, können wir einen lustigen *Ich-Ansatz* (*Objektansatz*) verwenden, der auch in der Analyse- und Designphase eingesetzt wird. Bei diesem Ich-Ansatz versetzen wir uns in das Objekt und sagen »Ich bin ...« für die Klasse, »Ich habe ...« für die Attribute und »Ich kann ...« für die Operationen. Meine Leserinnen und Leser sollten dies bitte an den Klassen *Mensch*, *Auto*, *Wurm* und *Kuchen* testen.

3.2.1 Klassenarbeit mit Point

Bevor wir uns mit eigenen Klassen beschäftigen, wollen wir zunächst einige Klassen aus der Standardbibliothek kennenlernen. Eine einfache Klasse ist *Point*. Sie beschreibt durch die Koordinaten *x* und *y* einen Punkt in einer zweidimensionalen Ebene und bietet einige Operationen an, mit denen sich Punkt-Objekte verändern lassen. Testen wir einen Punkt wieder mit dem Objektansatz:

Begriff	Erklärung
Klassenname	Ich bin ein Punkt .
Attribute	Ich habe eine x - und y -Koordinate.
Operationen	Ich kann mich verschieben und meine Position festlegen .

Tabelle 3.1 OOP-Begriffe und was sie bedeuten

Zu unserem Punkt können wir in der API-Dokumentation (<https://docs.oracle.com/en/java/javase/25/docs/api/java.desktop/java/awt/Point.html>) von Oracle nachlesen, dass er die Ob-

⁴ Den Begriff *Feld* benutze ich im Folgenden nicht. Er bleibt für Arrays reserviert.

jektvariablen `x` und `y` definiert, unter anderem eine Methode `setLocation(...)` besitzt und einen Konstruktor anbietet, der zwei Ganzzahlen annimmt.

3.3 Natürlich modellieren mit der UML (Unified Modeling Language) *

Für die Darstellung einer Klasse lässt sich Programmcode verwenden, also eine Textform, oder aber eine grafische Notation. Eine dieser grafischen Beschreibungsformen ist die UML. Grafische Abbildungen sind für Menschen deutlich besser zu verstehen und erhöhen die Übersicht.

Im ersten Abschnitt eines UML-Diagramms lassen sich die Attribute ablesen, im zweiten die Operationen. Das `+` vor den Eigenschaften (siehe Abbildung 3.1) zeigt an, dass sie öffentlich sind und jeder sie nutzen kann. Die Typangabe ist gegenüber Java umgekehrt: Zuerst kommt der Name der Variablen, dann der Typ bzw. bei Methoden der Typ des Rückgabewerts. Andere Programmiersprachen wie TypeScript oder Kotlin nutzen auch diese »umgedrehte« Typangabe im Code.

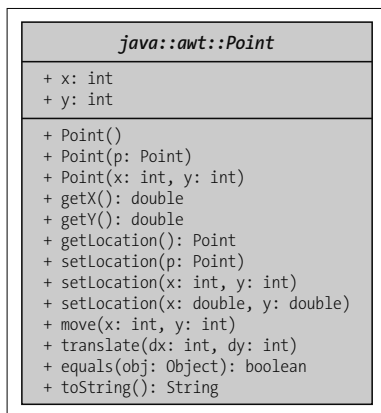


Abbildung 3.1 Die Klasse »java.awt.Point« in der UML-Darstellung

3.3.1 Wichtige Diagrammtypen der UML *

Die UML definiert diverse Diagrammtypen, die unterschiedliche Sichten auf die Software beschreiben können. Für die einzelnen Phasen im Softwareentwurf sind jeweils andere Diagramme wichtig. Wir wollen kurz vier Diagramme und ihre Einsatzgebiete besprechen.

Anwendungsfalldiagramm

Ein *Anwendungsfalldiagramm* (Use-Cases-Diagramm) entsteht meist während der Anforderungsphase und beschreibt die Geschäftsprozesse, indem es die Interaktion von Personen – oder von bereits existierenden Programmen – mit dem System darstellt. Die handelnden

Personen oder aktiven Systeme werden *Aktoren* genannt und sind im Diagramm als kleine (geschlechtslose) Männchen angedeutet. Anwendungsfälle (Use Cases) beschreiben dann eine Interaktion mit dem System.

Klassendiagramm

Für die statische Ansicht eines Programmentwurfs ist das *Klassendiagramm* einer der wichtigsten Diagrammtypen. Ein Klassendiagramm stellt zum einen die Elemente der Klasse dar, also die Attribute und Operationen, und zum anderen die Beziehungen der Klassen untereinander. Klassendiagramme werden in diesem Buch häufiger eingesetzt, um insbesondere die Assoziation und Vererbung zu anderen Klassen zu zeigen. Klassen werden in einem solchen Diagramm als Rechteck dargestellt, und die Beziehungen zwischen den Klassen werden durch Linien angedeutet.

Objektdiagramm

Ein Klassendiagramm und ein Objektdiagramm sind sich auf den ersten Blick sehr ähnlich. Der wesentliche Unterschied besteht darin, dass ein *Objektdiagramm* die Belegung der Attribute, also den Objektzustand, visualisiert. Dazu werden sogenannte *Ausprägungsspezifikationen* verwendet. Mit eingeschlossen sind die Beziehungen, die das Objekt zur Laufzeit mit anderen Objekten hält. Beschreibt zum Beispiel ein Klassendiagramm eine Person, so ist nur ein Rechteck im Diagramm. Hat diese Person zur Laufzeit Freunde (gibt es also Assoziationen zu anderen Person-Objekten), so können sehr viele Personen in einem Objektdiagramm verbunden sein, während ein Klassendiagramm diese Ausprägung nicht darstellen kann.

Sequenzdiagramm

Das *Sequenzdiagramm* stellt das dynamische Verhalten von Objekten dar. So zeigt es an, in welcher Reihenfolge Operationen aufgerufen und wann neue Objekte erzeugt werden. Die einzelnen Objekte bekommen eine vertikale Lebenslinie, und horizontale Linien zwischen den Lebenslinien der Objekte beschreiben die Operationen oder Objekterzeugungen. Das Diagramm liest sich somit von oben nach unten.

Da das Klassendiagramm und das Objektdiagramm eher die Struktur einer Software beschreiben, heißen die Modelle auch *Strukturdiagramme* (neben Paketdiagrammen, Komponentendiagrammen, Kompositionsstrukturdiagrammen und Verteilungsdiagrammen). Ein Anwendungsfalldiagramm und ein Sequenzdiagramm zeigen eher das dynamische Verhalten und werden *Verhaltensdiagramme* genannt. Weitere Verhaltensdiagramme sind das Zustandsdiagramm, das Aktivitätsdiagramm, das Interaktionsübersichtsdiagramm, das Kommunikationsdiagramm und das Zeitverlaufsdiagramm. In der UML ist es aber wichtig, die zentralen Aussagen des Systems in einem Diagramm festzuhalten, sodass sich problemlos Diagrammtypen mischen lassen.

In diesem Buch kommen fast nur Klassendiagramme vor.

3.4 Neue Objekte erzeugen

Eine Klasse beschreibt also, wie ein Objekt aussehen soll. In einer Mengen- bzw. Elementbeziehung ausgedrückt, entsprechen Objekte den Elementen und Klassen den Mengen, in denen die Objekte als Elemente enthalten sind. Diese Objekte haben Eigenschaften, die sich nutzen lassen. Wenn ein Punkt Koordinaten repräsentiert, wird es Möglichkeiten geben, diese Zustände zu erfragen und zu ändern.

Im Folgenden wollen wir untersuchen, wie sich von der Klasse `Point` zur Laufzeit Objekte erzeugen lassen und wie der Zugriff auf die Eigenschaften der `Point`-Objekte aussieht.

3.4.1 Ein Objekt einer Klasse mit dem Schlüsselwort `new` anlegen

Objekte müssen in Java immer ausdrücklich erzeugt werden. Dazu definiert die Sprache das Schlüsselwort `new`.

Beispiel

Die Java-Bibliothek deklariert für Punkte den Typ `Point`. Mit dem folgenden Code wird ein `Point`-Objekt erstellt:

```
new java.awt.Point();
```



Im Grunde ist `new` so etwas wie ein unärer Operator. Hinter dem Schlüsselwort `new` folgt der Name der Klasse, von der ein Objekt erzeugt werden soll. Der Klassenname ist hier voll qualifiziert angegeben, da sich `Point` in einem Paket `java.awt` befindet. (Ein Paket ist eine Gruppe zusammengehöriger Klassen; wir werden in Abschnitt 3.6.3, »Volle Qualifizierung und `import`-Deklaration«, sehen, dass diese Schreibweise auch abgekürzt werden kann.) Hinter dem Klassennamen folgt ein Paar runder Klammern für den *Konstruktoraufruf*. Dieser ist eine Art Methodenaufruf, über den sich Werte für die Initialisierung des frischen Objekts übergeben lassen.

Konnte die Speicherverwaltung von Java für das anzulegende Objekt freien Speicher reservieren und konnte der Konstruktor gültig durchlaufen werden, gibt der `new`-Ausdruck anschließend eine *Referenz* auf das frische Objekt an das Programm zurück. Merken wir uns diese Referenz nicht, kann die automatische Speicherbereinigung das Objekt wieder freigeben.

3.4.2 Deklarieren von Referenzvariablen

Das Ergebnis eines `new` ist eine Referenz auf das neue Objekt. Die Referenz wird in der Regel in einer *Referenzvariablen* zwischengespeichert, um später Eigenschaften des Objekts ansprechen zu können.

[zB]**Beispiel**

Deklariere die Variable `p` vom Typ `java.awt.Point`. Die Variable `p` nimmt anschließend die Referenz von dem neuen Objekt auf, das mit `new` angelegt wurde.

```
java.awt.Point p;
p = new java.awt.Point();
```

Die Deklaration und die Initialisierung einer Referenzvariablen lassen sich kombinieren (auch eine lokale Referenzvariable ist wie eine lokale Variable primitiven Typs zu Beginn uninitialisiert):

```
java.awt.Point p = new java.awt.Point();
```

Die Typen müssen natürlich kompatibel sein, und ein `Point`-Objekt geht nicht als `String` durch. Der Versuch, ein `Point`-Objekt einer `int`- oder `String`-Variablen zuzuweisen, ergibt somit einen Compilerfehler:

```
int    p = new java.awt.Point(); // ☠ Type mismatch: cannot convert from
                                   // Point to int
String s = new java.awt.Point(); // ☠ Type mismatch: cannot convert from
                                   // Point to String
```

Damit speichert eine Variable entweder einen einfachen Wert (Variable vom Typ `int`, `boolean`, `double` ...) oder einen Verweis auf ein Objekt. Der Verweis ist letztendlich intern ein Pointer auf einen Speicherbereich, doch der ist für Java-Entwickler so nicht sichtbar.

Referenztypen gibt es in vier Ausführungen: *Klassentypen*, *Schnittstellentypen* (auch *Interface-Typen* genannt), *Array-Typen* (auch *Feldtypen* genannt) und *Typvariablen* (eine Spezialisierung von generischen Typen). In unserem Fall haben wir ein Beispiel für einen Klassentyp.

IJ

Abbildung 3.2 Die Tastenkombination `Alt` + `↵` ermöglicht es, eine Variable für den Ausdruck anzulegen.

3.4.3 Jetzt mach mal 'nen Punkt: Zugriff auf Objektvariablen und -methoden

Die in einer Klasse deklarierten Variablen heißen *Objektvariablen* bzw. *Exemplar*-, *Instanz*- oder *Ausprägungsvariablen*. Jedes erzeugte Objekt hat seinen eigenen Satz von Objektvariablen:⁵ Sie bilden den Zustand des Objekts.

⁵ Es gibt auch den Fall, dass sich mehrere Objekte eine Variable teilen, sogenannte *statische Variablen*. Diesen Fall werden wir später in Kapitel 6, »Eigene Klassen schreiben«, genauer betrachten.

Der Punkt-Operator `.` erlaubt auf Objekten den Zugriff auf die Zustände oder den Aufruf von Methoden. Der Punkt steht zwischen einem Ausdruck, der eine Referenz liefert, und der Objekteigenschaft. Welche Eigenschaften eine Klasse genau bietet, zeigt die API-Dokumentation – wenn ein Objekt eine Eigenschaft nicht hat, wird der Compiler eine Nutzung verbieten.

Beispiel

Die Variable `p` referenziert ein `java.awt.Point`-Objekt. Die Objektvariablen `x` und `y` sollen initialisiert werden:

```
java.awt.Point p = new java.awt.Point();
p.x = 1;
p.y = 2 + p.x;
```

Ein Methodenaufruf gestaltet sich genauso einfach wie ein Zugriff auf Klassen- oder Objektvariablen. Hinter dem Ausdruck mit der Referenz folgt nach dem Punkt der Methodenname.

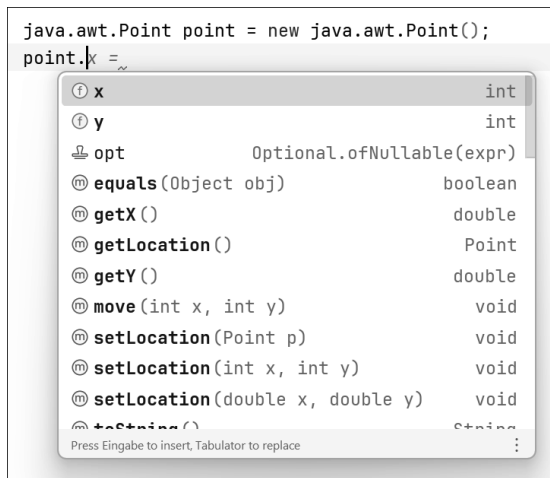


Abbildung 3.3 Die Tastenkombination `[Strg]` + Leertaste zeigt an, welche Eigenschaften eine Referenz ermöglicht. Eine Auswahl mit der `[↵]`-Taste wählt die Eigenschaft aus und setzt insbesondere bei Methoden den Cursor zwischen das Klammerpaar.

Tür und Spieler auf dem Spielbrett

Punkt-Objekte erscheinen auf den ersten Blick als mathematische Konstrukte, doch sie sind allgemein nutzbar. Alles, was eine Position im zweidimensionalen Raum hat, lässt sich gut durch ein Punkt-Objekt repräsentieren. Der Punkt speichert für uns `x` und `y`, und hätten wir keine Punkt-Objekte, so müssten wir `x` und `y` immer extra speichern.

Nehmen wir an, wir wollen einen Spieler und eine Tür auf ein Spielbrett setzen. Natürlich haben die beiden Objekte Positionen. Ohne Objekte würde eine Speicherung der Koordinaten vielleicht so aussehen:

[zB]

eclipse

```
int playerX;  
int playerY;  
int doorX;  
int doorY;
```

Die Modellierung ist nicht optimal, da wir mit der Klasse `Point` eine viel bessere Abstraktion haben, die zudem hübsche Methoden anbietet.

Ohne Abstraktion, nur die nackten Daten	Kapselung der Zustände in ein Objekt
<pre>int playerX; int playerY;</pre>	<pre>java.awt.Point player;</pre>
<pre>int doorX; int doorY;</pre>	<pre>java.awt.Point door;</pre>

Tabelle 3.2 Objekte kapseln Zustände.

Das folgende Beispiel erzeugt zwei Punkte, die die x/y-Koordinate eines Spielers und einer Tür auf einem Spielbrett repräsentieren. Nachdem die Punkte erzeugt wurden, werden die Koordinaten gesetzt, und es wird außerdem getestet, wie weit der Spieler und die Tür voneinander entfernt sind:

Listing 3.1 `src/main/java/PlayerAndDoorAsPoints.java`

```
static void main() {  
    java.awt.Point player = new java.awt.Point();  
    player.x = player.y = 10;  
  
    java.awt.Point door = new java.awt.Point();  
    door.setLocation( 10, 100 );  
  
    IO.println( player.distance( door ) ); // 90.0  
}
```

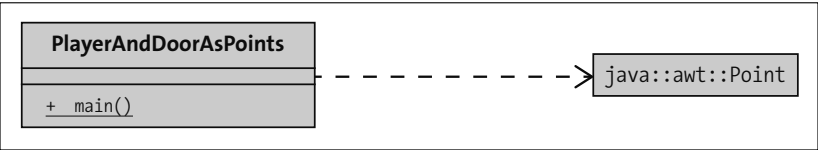


Abbildung 3.4 Die Abhängigkeit zwischen einer Klasse und dem »`java.awt.Point`« zeigt das UML-Diagramm mit einer gestrichelten Linie an. Attribute und Operationen von »`Point`« sind nicht dargestellt.

Im ersten Fall belegen wir die Variablen `x`, `y` des Spiels explizit. Im zweiten Fall setzen wir nicht direkt die Objektzustände über die Variablen, sondern verändern die Zustände über die Methode `setLocation(...)`. Die beiden Objekte besitzen eigene Koordinaten und kommen sich nicht in die Quere.

`toString()`

Die Methode `toString()` liefert als Ergebnis ein `String`-Objekt, das den Zustand des Punkts preisgibt. Sie ist insofern besonders, als es immer auf jedem Objekt eine `toString()`-Methode gibt – nicht in jedem Fall ist die Ausgabe allerdings sinnvoll.

Listing 3.2 `src/main/java/PointToStringDemo.java`

```
static void main() {
    java.awt.Point player = new java.awt.Point();
    java.awt.Point door   = new java.awt.Point();
    door.setLocation( 10, 100 );

    IO.println( player.toString() ); // java.awt.Point[x=0,y=0]
    IO.println( door );              // java.awt.Point[x=10,y=100]
}
```

Tipp

Anstatt für die Ausgabe explizit `println(obj.toString())` aufzurufen, funktioniert auch ein `println(obj)`. Das liegt daran, dass die Signatur `println(Object)` jedes beliebige Objekt als Argument akzeptiert und auf diesem Objekt automatisch die `toString()`-Methode aufruft.



Nach dem Punkt geht's weiter

Die Methode `toString()` liefert, wie wir gesehen haben, als Ergebnis ein `String`-Objekt:

```
java.awt.Point p = new java.awt.Point();
String s = p.toString();
IO.println( s ); // java.awt.Point[x=0,y=0]
```

Das `String`-Objekt besitzt selbst wieder Methoden. Eine davon ist `length()`, die die Länge der Zeichenkette liefert:

```
IO.println( s.length() ); // 23
```

Das Erfragen des `String`-Objekts und seiner Länge können wir zu einer Anweisung verbinden; wir sprechen von *kaskadierten Aufrufen*.

```
java.awt.Point p = new java.awt.Point();
IO.println( p.toString().length() ); // 23
```

Objekterzeugung ohne Variablenzuweisung

Bei der Nutzung von Objekteigenschaften muss der Typ links vom Punkt immer eine Referenz sein. Ob die Referenz nun aus einer Variablen kommt oder on-the-fly erzeugt wird, ist egal. Damit folgt, dass

```
java.awt.Point p = new java.awt.Point();
IO.println( p.toString().length() );           // 23
```

genau das Gleiche bewirkt wie:

```
IO.println( new java.awt.Point().toString().length() ); // 23
```

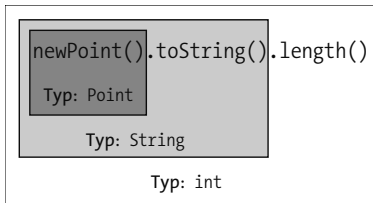


Abbildung 3.5 Jede Schachtelung ergibt einen neuen Typ.

Im Prinzip funktioniert auch Folgendes:

```
new java.awt.Point().x = 1;
```

Dies ist aber unsinnig, da zwar das Objekt erzeugt und eine Objektvariable gesetzt wird, anschließend das Objekt aber für die automatische Speicherbereinigung wieder Freiwild ist.



Beispiel

Finde über ein `File`-Objekt heraus, wie groß eine Datei ist:

```
long size = new java.io.File( "file.txt" ).length();
```

Die Rückgabe der `File`-Methode `length()` ist die Länge der Datei in Bytes.

3.4.4 Überblick über Point-Methoden

Ein paar Methoden der Klasse `Point` kamen schon vor, und die API-Dokumentation zählt selbstverständlich alle Methoden auf. Die interessanteren sind:

```
class java.awt.Point
extends Point2D
```

- `double getX()`
- `double getY()`
Liefert die x- bzw. y-Koordinate.

- `void setLocation(double x, double y)`
Setzt gleichzeitig die x- und die y-Koordinate. Die Koordinaten werden gerundet und in Ganzzahlen gespeichert.
- `boolean equals(Object obj)`
Prüft, ob ein anderer Punkt die gleichen Koordinaten besitzt. Dann ist die Rückgabe `true`, sonst `false`. Wird etwas anderes als ein `Point` übergeben, so wird der Compiler das nicht bemäkeln, nur wird das Ergebnis dann immer `false` sein.

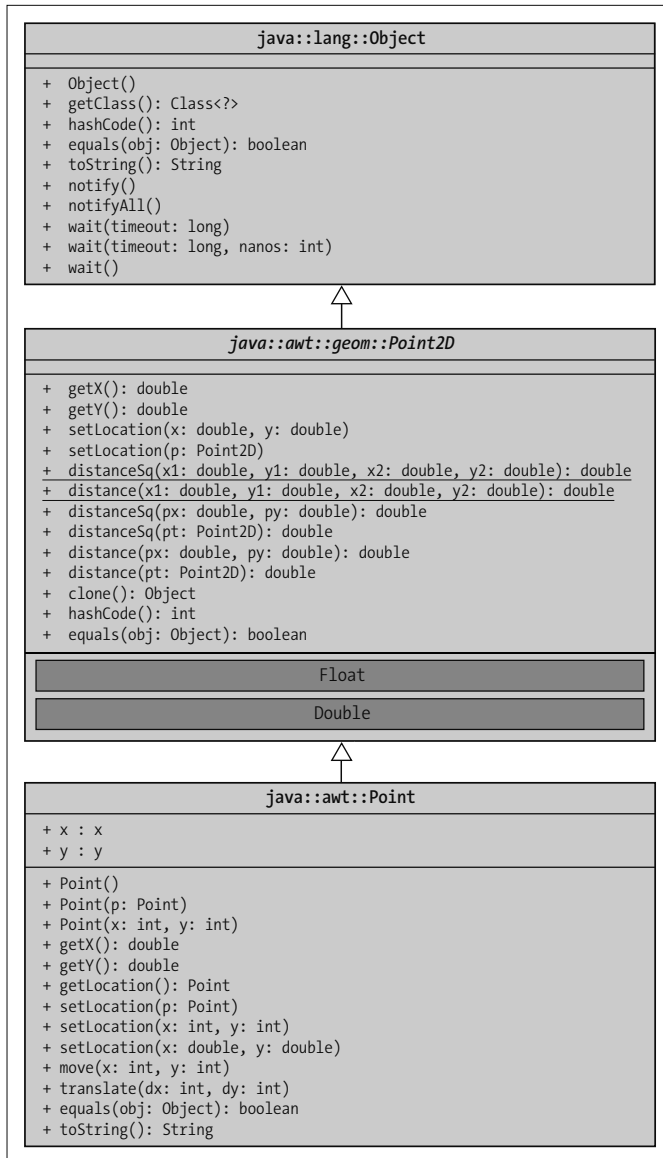


Abbildung 3.6 Vererbungshierarchie bei Point2D



Hinweis

Es ist überraschend, dass ein `Point` die Koordinaten als `int` speichert, aber die Methoden `getX()` und `getY()` ein `double` liefern und `setLocation(double, double)` die Koordinaten als `double` annimmt, rundet und als `int` ablegt, also Genauigkeit verliert. Der Grund hat etwas mit Vererbung zu tun, was in Kapitel 7 ausführlicher beleuchtet wird. `Point` erbt von `Point2D`, und dort gibt es schon `double getX()`, `double getY()` und `setLocation(double, double)`; die Unterklasse `Point` kann nicht einfach aus `double` ein `int` machen.

Ein paar Worte über Vererbung und die API-Dokumentation *

Eine Klasse besitzt nicht nur eigene Eigenschaften, sondern erbt auch immer welche von ihren Eltern. Im Fall von `Point` ist die Oberklasse `Point2D` – so sagt es die API-Dokumentation. Selbst `Point2D` erbt von `Object`, einer magischen Klasse, die alle Java-Klassen als Oberklasse haben. Der Vererbung widmen wir später das sehr ausführliche Kapitel 7, »Objektorientierte Beziehungsfragen«, aber es ist jetzt schon wichtig zu verstehen, dass die Oberklasse Objektvariablen und Methoden an Unterklassen weitergibt. Sie sind in der API-Dokumentation einer Klasse nur kurz im Block »Methods inherited from ...« aufgeführt und gehen schnell unter. Es ist unabdingbar, beim Entwickeln nicht nur bei den Methoden der Klasse selbst zu schauen, sondern auch bei den geerbten Methoden. Bei `Point` sind es also nicht nur die Methoden dort selbst, sondern auch die Methoden aus `Point2D` und `Object`.

Nehmen wir uns einige Methoden der Oberklasse vor. Die Klassendeklaration von `Point` enthält ein `extends Point2D`, was explizit klarmacht, dass es eine Oberklasse gibt:⁶

```
class java.awt.Point
    extends Point2D
```

- `static double distance(double x1, double y1, double x2, double y2)`
Berechnet den Abstand zwischen den gegebenen Punkten nach der euklidischen Distanz.
- `double distance(double x, double y)`
Berechnet den Abstand des aktuellen Punkts zu angegebenen Koordinaten.
- `double distance(Point2D pt)`
Berechnet den Abstand des aktuellen Punkts zu den Koordinaten des übergebenen Punkts.

⁶ Damit ist die Klassendeklaration noch nicht vollständig, da ein `implements Serializable` fehlt, doch das soll uns jetzt erst einmal egal sein.

Sind zwei Punkte gleich?

Ob zwei Punkte gleichwertig sind, sagt uns die `equals(...)`-Methode. Die Anwendung ist einfach. Stellen wir uns vor, wir wollen Koordinaten für einen Spieler, eine Tür und eine Schlange verwalten und dann testen, ob der Spieler »auf« der Tür steht und die Schlange auf der Position des Spielers:

Listing 3.3 `src/main/java/PointEqualsDemo.java`

```
static void main() {
    java.awt.Point player = new java.awt.Point();
    player.x = player.y = 10;

    java.awt.Point door = new java.awt.Point();
    door.setLocation( 10, 10 );

    IO.println( player.equals( door ) );    // true
    IO.println( door.equals( player ) );    // true

    java.awt.Point snake = new java.awt.Point();
    snake.setLocation( 20, 22 );

    IO.println( snake.equals( door ) );    // false
}
```

Da Spieler und Tür die gleichen Koordinaten (10, 10) besitzen, liefert `equals(...)` die Rückgabe `true`. Ob wir den Abstand vom Spieler zur Tür berechnen lassen oder den Abstand von der Tür zum Spieler – das Ergebnis bei `equals(...)` sollte immer symmetrisch sein. Die Schlange befindet sich an einer anderen Position (20, 22), weshalb der Vergleich mit der Tür `false` ergibt.

Eine andere Testmöglichkeit ergibt sich durch `distance(...)`, denn ist der Abstand der Punkte null, so liegen die Punkte natürlich aufeinander und haben keinen Abstand.

Listing 3.4 `src/main/java/Distances.java`

```
static void main() {
    java.awt.Point player = new java.awt.Point();
    player.setLocation( 10, 10 );
    java.awt.Point door = new java.awt.Point();
    door.setLocation( 10, 10 );
    java.awt.Point snake = new java.awt.Point();
    snake.setLocation( 20, 10 );

    IO.println( player.distance( door ) );    // 0.0
    IO.println( player.distance( snake ) );    // 10.0
}
```

```
10.println( player.distance( snake.x, snake.y ) ); // 10.0
}
```

Spieler, Tür und Schlange sind wieder als `Point`-Objekte repräsentiert und mit Positionen vorgelegt. Beim `player` rufen wir die Methode `distance(...)` auf und übergeben den Verweis auf die Tür und die Schlange.

3.4.5 Konstruktoren nutzen

Werden Objekte mit `new` angelegt, so wird ein Konstruktor aufgerufen. Ein Konstruktor hat die Aufgabe, ein Objekt in einen Startzustand zu versetzen, zum Beispiel die Objektvariablen zu initialisieren. Ein Konstruktor ist dazu ein guter Weg, denn er wird immer als Erstes aufgerufen, noch bevor eine andere Methode aufgerufen wird. Die Initialisierung im Konstruktor stellt sicher, dass das neue Objekt einen sinnvollen Anfangszustand aufweist.

Aus der API-Dokumentation von `Point` sind drei Konstruktoren abzulesen:

```
class java.awt.Point
extends Point2D
```

- `Point()`
Legt einen Punkt mit den Koordinaten (0, 0) an.
- `Point(int x, int y)`
Legt einen neuen Punkt an und initialisiert ihn mit den Werten aus `x` und `y`.
- `Point(Point p)`
Legt einen neuen Punkt an und initialisiert ihn mit den gleichen Koordinaten, die der übergebene Punkt hat. Wir nennen so einen Konstruktor auch *Copy-Konstruktor*.

Ein Konstruktor ohne Argumente ist der *parameterlose Konstruktor*, selten auch *No-Arg-Konstruktor* genannt. Jede Klasse kann höchstens einen parameterlosen Konstruktor besitzen, es kann aber auch sein, dass eine Klasse keinen parameterlosen Konstruktor deklariert, sondern nur Konstruktoren mit Parametern, also parametrisierte Konstruktoren.

[zB]

Beispiel

Die drei folgenden Varianten legen ein `Point`-Objekt mit denselben Koordinaten (1, 2) an; `java.awt.Point` ist mit `Point` abgekürzt:

```
Point p = new Point(); p.setLocation( 1, 2 );
Point q = new Point( 1, 2 );
Point r = new Point( q );
```

Als Erstes steht der parameterlose Konstruktor, im zweiten und dritten Fall handelt es sich um parametrisierte Konstruktoren.

3.5 ZZZZZnake

Ein Klassiker aus dem Genre der Computerspiele ist *Snake*. Auf dem Bildschirm gibt es den Spieler, eine Schlange, Gold und eine Tür. Die Tür und das Gold sind fest, den Spieler können wir bewegen, und die Schlange bewegt sich selbstständig auf den Spieler zu. Wir müssen versuchen, die Spielfigur zum Gold zu bewegen und dann zur Tür. Wenn die Schlange uns vorher erwischt, haben wir Pech gehabt, und das Spiel ist verloren.

Vielleicht hört sich das auf den ersten Blick komplex an, aber wir haben alle Bausteine zusammen, um dieses Spiel zu programmieren:

- ▶ Spieler, Schlange, Gold und Tür sind `Point`-Objekte, die mit Koordinaten vorkonfiguriert sind.
- ▶ Eine Schleife läuft alle Koordinaten ab. Ist ein Spieler, die Tür, die Schlange oder Gold »getroffen«, gibt es eine symbolische Darstellung der Figuren.
- ▶ Wir testen drei Bedingungen für den Spielstatus: 1. Hat der Spieler das Gold eingesammelt und steht auf der Tür? (Das Spiel ist zu Ende.) 2. Beißt die Schlange den Spieler? (Das Spiel ist verloren.) 3. Sammelt der Spieler Gold ein?
- ▶ Mit dem `Scanner` können wir auf Tastendrücke reagieren und den Spieler auf dem Spielbrett bewegen.
- ▶ Die Schlange muss sich in Richtung des Spielers bewegen. Während der Spieler sich nur entweder horizontal oder vertikal bewegen kann, erlauben wir der Schlange, sich diagonal zu bewegen.

Im Quellcode sieht das so aus:

Listing 3.5 `src/main/java/ZZZZZnake.java`

```
static void main() {
    java.awt.Point playerPosition = new java.awt.Point( 10, 9 );
    java.awt.Point snakePosition  = new java.awt.Point( 30, 2 );
    java.awt.Point goldPosition   = new java.awt.Point( 6, 6 );
    java.awt.Point doorPosition   = new java.awt.Point( 0, 5 );
    boolean rich = false;

    while ( true ) {
        // Draw grid and symbols

        for ( int y = 0; y < 10; y++ ) {
            for ( int x = 0; x < 40; x++ ) {
                java.awt.Point p = new java.awt.Point( x, y );
                if ( playerPosition.equals( p ) )
                    IO.print( '&' );
                else if ( snakePosition.equals( p ) )
```

```
        IO.print( 'S' );
    else if ( goldPosition.equals( p ) )
        IO.print( '$' );
    else if ( doorPosition.equals( p ) )
        IO.print( '#' );
    else IO.print( '.' );
    }
    IO.println();
}

// Determine status

if ( rich && playerPosition.equals( doorPosition ) ) {
    IO.println( "You won!" );
    return;
}
if ( playerPosition.equals( snakePosition ) ) {
    IO.println( "SSSSS. You were bitten by the snake!" );
    return;
}
if ( playerPosition.equals( goldPosition ) ) {
    rich = true;
    goldPosition.setLocation( -1, -1 );
}

// Console input and change player position
// Keep playing field between 0/0.. 39/9
switch ( new java.util.Scanner( System.in ).next() ) {
    case "u" /* p */ -> playerPosition.y = Math.max( 0, playerPosition.y - 1 );
    case "d" /* own */ -> playerPosition.y = Math.min( 9, playerPosition.y + 1 );
    case "l" /* eft */ -> playerPosition.x = Math.max( 0, playerPosition.x - 1 );
    case "r" /* ight */ -> playerPosition.x = Math.min( 39, playerPosition.x + 1 );
}

// Snake moves towards the player

if ( playerPosition.x < snakePosition.x )
    snakePosition.x--;
else if ( playerPosition.x > snakePosition.x )
    snakePosition.x++;
if ( playerPosition.y < snakePosition.y )
    snakePosition.y--;
else if ( playerPosition.y > snakePosition.y )
```

```
        snakePosition.y++;  
    } // end while  
}
```

Die Point-Eigenschaften, die wir nutzen, sind:

- ▶ Die Objektzustände `x`, `y`: Der Spieler und die Schlange werden bewegt, und die Koordinaten müssen neu gesetzt werden.
- ▶ Die Methode `setLocation(...)`: Ist das Gold aufgesammelt, setzen wir die Koordinaten so, dass die Koordinate vom Gold nicht mehr auf unserem Raster liegt.
- ▶ Die Methode `equals(...)`: Testet, ob ein Punkt auf einem anderen Punkt steht.

3.5.1 Erweiterung

Wer Lust hat, an der Aufgabe noch ein wenig weiterzuprogrammieren, der kann Folgendes tun:

- ▶ Spieler, Schlange, Gold und Tür sollen auf Zufallskordinaten gesetzt werden.
- ▶ Statt nur eines Stücks Gold soll es zwei Stücke geben.
- ▶ Statt einer Schlange soll es zwei Schlangen geben.
- ▶ Mit zwei Schlangen und zwei Stücken Gold kann es etwas eng für den Spieler werden. Er soll daher am Anfang fünf Züge machen können, ohne dass die Schlangen sich bewegen.
- ▶ Für Vorarbeiter: Das Programm, das sich bisher nur in der `main`-Methode befindet, soll in verschiedene Methoden aufgespalten werden.

3.6 Pakete schnüren, Importe und Compilationseinheiten

Die Java-Klassenbibliothek umfasst Tausende von Typen und bietet damit eine breite Grundlage für plattformunabhängige Programme. Sie enthält unter anderem Datenstrukturen, Klassen zur Datums- und Zeitberechnung sowie zur Dateiverarbeitung. Die meisten Typen sind in Java selbst implementiert, und der Quellcode ist in der Regel direkt aus der Entwicklungsumgebung zugänglich. Einige Komponenten greifen jedoch auf native Implementierungen zurück, beispielsweise beim Lesen aus Dateien.

Wenn wir eigene Klassen programmieren, ergänzen sie sozusagen die Standardbibliothek; im Endeffekt wächst damit die Anzahl der möglichen Typen, die ein Programm nutzen kann.

3.6.1 Java-Pakete

Ein *Paket* ist eine Gruppe thematisch zusammengehöriger Typen. Pakete lassen sich in Hierarchien ordnen, sodass ein Paket wieder ein anderes Paket enthalten kann; das ist genauso wie bei der Verzeichnisstruktur des Dateisystems. Beispiele für Pakete sind:

- ▶ `java.awt`
- ▶ `java.util`
- ▶ `com.google`
- ▶ `org.apache.commons.math3.fraction`
- ▶ `com.tutego.insel`

Die Klassen der Java-Standardbibliothek befinden sich in Paketen, die mit `java` und `javax` beginnen. Google nutzt die Wurzel `com.google`; die Apache Foundation veröffentlicht Java-Code unter `org.apache`. So können wir von außen ablesen, von welchen Typen die eigene Klasse abhängig ist.

3.6.2 Pakete der Standardbibliothek

Die logische Gruppierung und Hierarchie lässt sich sehr gut an der Java-Bibliothek beobachten. Die Java-Standardbibliothek beginnt mit der Wurzel `java`, einige Typen liegen in `javax`. Unter diesem Paket liegen weitere Pakete, etwa `awt`, `math` und `util`. In `java.math` liegen zum Beispiel die Klassen `BigInteger` und `BigDecimal`, denn die Arbeit mit beliebig großen Ganzzahl- und Dezimalzahlen gehört eben zum Mathematischen. Ein Punkt und ein Polygon, repräsentiert durch die Klassen `Point` und `Polygon`, gehören in das Paket für grafische Oberflächen, und das ist das Paket `java.awt`.

Wenn jemand eigene Klassen in Pakete mit dem Präfix `java` setzen würde, etwa `java.tutego`, würde ein Programmautor damit Verwirrung stiften, da nicht mehr nachvollziehbar ist, ob das Paket Bestandteil jeder Distribution ist. Daher ist dieses Präfix für eigene Pakete verboten.

Klassen, die in einem Paket liegen, das mit `javax` beginnt, können Teil der Java SE sein wie zum Beispiel `javax.swing`, müssen aber nicht zwingend zur Java SE gehören; dazu folgt mehr in Abschnitt 16.1.2, »Übersicht über die Pakete der Standardbibliothek«.

3.6.3 Volle Qualifizierung und import-Deklaration

Um die Klasse `Point`, die im Paket `java.awt` liegt, außerhalb des Pakets `java.awt` zu nutzen – und das ist für uns als Nutzende immer der Fall –, muss sie dem Compiler mit der gesamten Paketangabe bekannt gemacht werden. Hierzu reicht der Klassenname allein nicht aus, denn es kann sein, dass der Klassenname mehrdeutig ist und eine Klassendeklaration in unterschiedlichen Paketen existiert.

Typen sind erst durch die Angabe ihres Pakets eindeutig identifiziert. Ein Punkt trennt Pakete, also schreiben wir `java.awt` und `java.util` – nicht einfach nur `awt` oder `util`. Mit einer

weltweit unzähligen Anzahl von Paketen und Klassen wäre sonst eine Eindeutigkeit gar nicht machbar. Es kann in verschiedenen Paketen durchaus ein Typ mit gleichem Namen vorkommen, etwa `java.util.List` und `java.awt.List` oder `java.util.Date` und `java.sql.Date`. Daher bilden nur Paket und Typ zusammen eine eindeutige Kennung.

Um dem Compiler die präzise Zuordnung einer Klasse zu einem Paket zu ermöglichen, gibt es zwei Möglichkeiten: Zum einen lassen sich die Typen voll qualifizieren, wie wir das bisher getan haben. Eine alternative und praktischere Möglichkeit besteht darin, den Compiler mit einer `import`-Deklaration auf die Typen im Paket aufmerksam zu machen:

Volle Qualifikation	import-Deklaration
Listing 3.6 AwtWithoutImport.java <pre> static void main(){ java.awt.Point p = new java.awt.Point(); java.awt.Polygon t = new java.awt.Polygon(); t.addPoint(10, 10); t.addPoint(10, 20); t.addPoint(20, 10); IO.println(p); IO.println(t.contains(15,15)); } </pre>	Listing 3.7 AwtWithImport.java <pre> import java.awt.Point; import java.awt.Polygon; static void main(){ Point p = new Point(); Polygon t = new Polygon(); t.addPoint(10, 10); t.addPoint(10, 20); t.addPoint(20, 10); IO.println(p); IO.println(t.contains(15,15)); } </pre>

Tabelle 3.3 Typzugriff über volle Qualifikation und mit »import«-Deklaration

Während der Quellcode auf der linken Seite die volle Qualifizierung verwendet und jeder Verweis auf einen Typ mehr Schreibarbeit kostet, ist im rechten Fall bei der `import`-Deklaration nur der Klassenname genannt und die Paketangabe in ein `import` »ausgelagert«. Alle Typen, die bei `import` genannt werden, merkt sich der Compiler für diese Datei in einer Datenstruktur. Kommt der Compiler zu der Zeile mit `Point p = new Point();`, findet er den Typ `Point` in seiner Datenstruktur und kann den Typ dem Paket `java.awt` zuordnen. Damit ist wieder die unabkömmliche Qualifizierung gegeben.

**Hinweis**

Die Typen aus `java.lang` sind automatisch importiert, sodass z. B. ein `import java.lang.String`; nicht nötig ist.

3.6.4 Mit `import p1.p2.*` alle Typen eines Pakets erreichen

Greift eine Java-Klasse auf mehrere andere Typen des gleichen Pakets zurück, kann die Anzahl der `import`-Deklarationen groß werden. In unserem Beispiel nutzen wir mit `Point` und `Polygon` nur zwei Klassen aus `java.awt`, aber es lässt sich schnell ausmalen, was passiert, wenn aus dem Paket für grafische Oberflächen zusätzlich Fenster, Beschriftungen, Schaltflächen, Schieberegler usw. eingebunden werden. In diesem Fall darf ein `*` als letztes Glied in einer `import`-Deklaration stehen:

```
import java.awt.*;
import java.math.*;
```

Mit dieser Syntax kennt der Compiler alle sichtbaren Typen in den Paketen `java.awt` und `java.math`, sodass der Compiler das Paket für die Klassen `Point` und `Polygon` zuordnen kann sowie auch das Paket für die Klasse `BigInteger`.

**Hinweis**

Das `*` ist nur auf der letzten Hierarchieebene erlaubt und gilt immer für alle Typen in diesem Paket. Syntaktisch falsch sind:

```
import *; // ☠ Syntax error on token "*", Identifier expected
import java.awt.Po*; // ☠ Syntax error on token "*", delete this token
```

Eine Anweisung wie `import java.*`; ist zwar syntaktisch korrekt, aber dennoch ohne Wirkung, denn direkt im Paket `java` gibt es keine Typdeklarationen, sondern nur Unterpakete.

Die `import`-Deklaration bezieht sich nur auf ein Verzeichnis (in der Annahme, dass die Pakete auf das Dateisystem abgebildet werden) und schließt die Unterverzeichnisse nicht ein.

Das `*` verkürzt zwar die Anzahl der individuellen `import`-Deklarationen, es ist aber gut, zwei Dinge im Kopf zu behalten:

- Falls zwei unterschiedliche Pakete einen gleichlautenden Typ beherbergen, etwa `Date` in `java.util` und `java.sql` oder `List` in `java.awt` und `java.util`, so kommt es bei der Verwendung des Typs zu einem Übersetzungsfehler, weil der Compiler nicht weiß, was gemeint ist. Eine volle Qualifizierung löst das Problem.

- Die Anzahl der `import`-Deklarationen sagt etwas über den Grad der Komplexität aus. Je mehr `import`-Deklarationen es gibt, desto größer werden die Abhängigkeiten zu anderen Klassen, was im Allgemeinen ein Alarmzeichen ist. Zwar zeigen grafische Tools die Abhängigkeiten genau an, doch ein `import *` kann diese erst einmal verstecken.

Best Practice

Entwicklungsumgebungen setzen die `import`-Deklarationen in der Regel automatisch und falten die Blöcke üblicherweise ein. Daher sollte der `*` nur sparsam eingesetzt werden, denn er »verschmutzt« den Namensraum durch viele Typen und erhöht die Gefahr von Kollisionen.



3.6.5 Modul-Import

In größeren Java-Programmen sammeln sich naturgemäß zahlreiche `import`-Deklarationen an. Zwar lassen sich diese in modernen Entwicklungsumgebungen bequem einklappen, doch sie bleiben Teil des Quellcodes und können schnell unübersichtlich werden.

Java bietet einige Möglichkeiten, die Importe zu vereinfachen. Eine Option ist der Einsatz von Wildcards – etwa `import java.awt.*` –, um sämtliche Typen eines Pakets auf einmal zu importieren. Außerdem wird das Paket `java.lang` automatisch importiert, sodass dort Typen wie `String`, `Math` oder `Comparable` ganz ohne explizite `import`-Deklaration verwendet werden können.

In Java 25 wurden *Modul-Importe* eingeführt, die alle öffentlichen Typen der exportierten Pakete eines Moduls sowie die Typen der indirekt referenzierten Module einbinden.

Erklärung

Ein Paket ist eine Sammlung thematisch zusammengehöriger Typen – wie Klassen und Schnittstellen –, die gemeinsam eine bestimmte Funktionalität bereitstellen. Ein Modul ist eine übergeordnete Einheit, die mehrere Pakete logisch zusammenfasst. Module dienen der besseren Strukturierung größerer Anwendungen und ermöglichen es, explizit festzulegen, welche Teile eines Moduls für andere Module zugänglich sind und von welchen Modulen es selbst abhängt. Die Java-Standardbibliothek ist vollständig modularisiert. Eigene Module spielen in der Praxis eine untergeordnete Rolle.



Die Syntax eines Modul-Imports lautet wie folgt:

```
import module Modulname;
```

Ein typisches Beispiel ist der Import des Basismoduls der Java-Standardbibliothek:

```
import module java.base;
```

Das Modul `java.base` ist das zentrale Kernmodul und umfasst essenzielle Pakete wie `java.io`, `java.net`, `java.util` und weitere. Die Javadoc zeigt alle Module und ihre enthaltenen Pakete auf.

Namenskonflikte durch Mehrfach-Importe

Beim Einsatz von Wildcards in mehreren Paket-Importen kann es zu Namenskonflikten kommen, wenn gleichnamige Typen in unterschiedlichen Paketen vorhanden sind. Dasselbe Problem kann auch bei Modul-Importen auftreten, da ein Modul in der Regel viele Pakete mit potenziell gleichnamigen Typen enthält. Ein klassisches Beispiel sind die Typen `Date` und `List`, die in mehreren Paketen existieren: Es gibt `Date` in `java.util` (Modul `java.base`) und in `java.sql` (Modul `java.sql`) und `List` in `java.util` (Modul `java.base`) und in `java.awt` (Modul `java.desktop`).

Nehmen wir folgenden Modul-Import an:

```
import module java.base;    // Enthält java.util.Date und java.util.List
import module java.sql;     // Enthält java.sql.Date
import module java.desktop; // Enthält java.awt.List
```

Hier meldet der Compiler einen Fehler, wenn `Date` oder `List` im Programm verwendet werden, da `Date` entweder `java.util.Date` oder `java.sql.Date` bedeuten könnte und `List` entweder `java.util.List` oder `java.awt.List`. Der Konflikt tritt also auf, weil die Modul-Importe mehrere Pakete einbinden, die Typen mit denselben einfachen Namen exportieren.

Es gibt unterschiedliche Lösungsansätze für das Problem.

Lösung 1: Eine gezielte `import`-Deklaration für den gewünschten Typ verwenden:

```
import module java.base;
import module java.sql;
import module java.desktop;
import java.sql.Date;    // Eindeutig - nutzt Date aus java.sql und
import java.util.List;   // List aus dem Paket java.util
```

Lösung 2: Ein Wildcard-Import für die Priorisierung eines Pakets:

```
import module java.base;
import module java.sql;
import module java.desktop;
import java.util.*;    // Priorisiert Typen aus java.util gegenüber Modul-Importen
import java.sql.*;     // Priorisiert Typen aus java.sql gegenüber Modul-Importen
```

Die Mehrdeutigkeit für `Date` bleibt bestehen, da die Wildcard-Importe `import java.util.*` und `import java.sql.*` gleichrangig sind. Um `Date` eindeutig zu machen, ist ein gezielter Klassen-Import wie `import java.sql.Date` erforderlich. Der Typ `List` wird durch `import java.util.*` auf `java.util.List` festgelegt, da dieser Wildcard-Import den Modul-Import `java.desktop` überlagert.

Lösung 3: Vollständige Qualifikation von Typen:

```
import module java.base;
import module java.sql;
import module java.desktop;
```

Und dann zum Beispiel:

```
java.sql.Date sqlDate = new java.sql.Date(System.currentTimeMillis());
```

Um es zusammenzufassen: Die Regel der Spezifität hilft bei der Auflösung solcher Namenskonflikte:

- ▶ Einzelne Typ-Importe (`import java.sql.Date`) haben Vorrang vor
- ▶ Paket-Importen mit Wildcard (`import java.util.*`), die wiederum Vorrang haben vor
- ▶ Modul-Importen (`import module java.base`).

Strukturierung

Zur besseren Lesbarkeit des Codes empfiehlt es sich, `import`-Deklarationen logisch zu gruppieren, in folgender Reihenfolge:

```
// Modul-Importe
import module java.base;
import module java.sql;

// Paket-Importe
import java.util.*;
import javax.sql.*;

// Einzelne Typ-Importe
import java.sql.Date;

class Application { }
```

Automatischer Modul-Import in kompakten Java-Quelldateien

In kompakten Java-Quelldateien wird das Modul `java.base` automatisch importiert – so, als stünde am Anfang:

```
import module java.base;
```

Damit stehen alle öffentlichen Typen der vom Modul `java.base` exportierten Pakete automatisch und ohne expliziten `import` zur Verfügung. Dazu gehören z. B. Typen aus `java.util`, `java.io` oder `java.math`. Der Typ `java.awt.Point` gehört hingegen zum Modul `java.desktop` und steht daher *nicht* automatisch zur Verfügung, da dieses Modul nicht Teil von `java.base` ist.

3.6.6 Hierarchische Strukturen über Pakete und die Spiegelung im Dateisystem

Die Klassen eines Pakets befinden sich üblicherweise im gleichen Verzeichnis.⁷ Der Paketname entspricht dem Verzeichnisnamen und umgekehrt. Dabei ersetzt ein Punkt im Paketnamen den Verzeichnistrenner (»\« unter Windows bzw. »/« unter Unix).



Beispiel

Gegeben sei die Verzeichnisstruktur `com/tutego/insel/printer/DatePrinter.class` mit einer Hilfsklasse. Der zugehörige Paketname lautet `com.tutego.insel.printer` – entsprechend dem Verzeichnispfad `com/tutego/insel/printer`.

Der Aufbau von Paketnamen

Paketnamen können prinzipiell frei gewählt werden. In der Praxis richtet sich die Benennung jedoch häufig nach der umgekehrten Internetdomäne der Organisation. Aus der Domäne `tutego.com` wird somit das Paketpräfix `com.tutego`. Diese Konvention trägt dazu bei, Klassennamen weltweit eindeutig zu halten. Paketnamen werden dabei grundsätzlich kleingeschrieben. Umlaute und Sonderzeichen sollten in Paketnamen vermieden werden, da sie auf Dateisystemen häufig zu Problemen führen. Zudem gilt ohnehin die Konvention, Bezeichner in englischer Sprache zu wählen.

3.6.7 Die package-Deklaration

Um die Klasse `DatePrinter` in ein Paket `com.tutego.insel.printer` zu setzen, müssen zwei Bedingungen erfüllt sein:

- Die Datei muss sich physisch im gleichnamigen Verzeichnis befinden, also in `com/tutego/insel/printer`.
- Der Quellcode muss ganz oben eine `package`-Deklaration enthalten.

Steht die `package`-Deklaration nicht ganz am Anfang, gibt es einen Übersetzungsfehler (selbstverständlich lassen sich Kommentare vor die `package`-Deklaration setzen).

Hier der vollständige Code für die Klasse `DatePrinter`:

Listing 3.8 `src/main/java/com/tutego/insel/printer/DatePrinter.java`

```
package com.tutego.insel.printer;
```

```
import java.time.LocalDate;
```

⁷ Ich schreibe »üblicherweise«, da die Paketstruktur nicht zwingend auf Verzeichnisse abgebildet werden muss. Pakete könnten beispielsweise vom Klassenlader aus einer Datenbank gelesen werden. Im Folgenden wollen wir aber immer von Verzeichnissen ausgehen.

```
import java.time.format.*;

public class DatePrinter {
    public static void printCurrentDate() {
        var fmt = DateTimeFormatter.ofLocalizedDate( FormatStyle.MEDIUM );
        IO.println( LocalDate.now().format( fmt ) );
    }
}
```

Hinter die package-Deklaration kommen wie gewohnt import-Deklaration(en) und die Typdeklaration(en).

Hinweis

Kompakte Quelldateien dürfen keine package-Deklaration enthalten! Sie sind eigenständige Mini-Programme und nicht Teil eines größeren Programms. Es ist jedoch möglich, sie in Paketverzeichnissen abzulegen, um sie thematisch zu ordnen. Das hat keine Auswirkung auf ihre Paketzugehörigkeit im Code.



Klasse verwenden

Um die Klasse zu nutzen, bieten sich wie bekannt zwei Möglichkeiten: einmal über die volle Qualifizierung und einmal über die import-Deklaration.

Die erste Variante sieht so aus:

Listing 3.9 src/main/java/DatePrinterUser1.java

```
static void main() {
    com.tutego.insel.printer.DatePrinter.printCurrentDate();
}
```

Und hier ist die Variante mit der import-Deklaration:

Listing 3.10 src/main/java/DatePrinterUser2.java

```
import com.tutego.insel.printer.DatePrinter;

static void main() {
    DatePrinter.printCurrentDate();
}
```

**Tipp**

Eine Entwicklungsumgebung nimmt uns viel Arbeit ab, daher bemerken wir die Dateioperationen – wie das Anlegen von Verzeichnissen – in der Regel nicht. Auch das Verschieben von Typen in andere Pakete und die damit verbundenen Änderungen im Dateisystem sowie die Anpassungen an den `import`- und `package`-Deklarationen übernimmt eine moderne IDE für uns.

3.6.8 Unbenanntes Paket (default package)

Eine Klasse ohne Paketangabe befindet sich im *unbenannten Paket* (engl. *unnamed package*) bzw. *Default-Paket*. Es ist eine gute Idee, eigene Klassen immer in Paketen zu organisieren. Das erlaubt feinere Sichtbarkeiten und verhindert Konflikte mit Code aus anderen Quellen. Es wäre ein großes Problem, wenn a) jedes Unternehmen unübersichtlich alle Klassen in das unbenannte Paket setzen und dann b) versuchen würde, die Bibliotheken auszutauschen: Konflikte wären vorprogrammiert.

Eine im Paket befindliche Klasse kann jede andere sichtbare Klasse aus anderen Paketen importieren, aber keine Klassen aus dem unbenannten Paket. Nehmen wir `Sugar` im unbenannten Paket und `Chocolate` im Paket `com.tutego` an:

```
Sugar.class  
com/tutego/insel/Chocolate.class
```

Die Klasse `Chocolate` kann `Sugar` nicht nutzen, da Klassen aus dem unbenannten Paket nicht für Unterpakete sichtbar sind. Nur andere Klassen im unbenannten Paket können Klassen im unbenannten Paket nutzen.

Stünde nun `Sugar` in einem Paket – das auch ein Oberpaket sein kann! –, so wäre das wiederum möglich, und `Chocolate` könnte `Sugar` importieren:

```
com/Sugar.class  
com/tutego/insel/Chocolate.class
```

3.6.9 Compilationseinheit (Compilation Unit)

Eine `.java`-Datei ist eine *Compilationseinheit* (*Compilation Unit*), die aus drei (optionalen) Segmenten besteht – in dieser Reihenfolge:

1. `package`-Deklaration
2. `import`-Deklaration(en)
3. Typdeklaration(en)

So besteht eine *Compilationseinheit* aus höchstens einer Paketdeklaration (nicht nötig, wenn der Typ im Default-Paket stehen soll), beliebig vielen `import`-Deklarationen und belie-

big vielen Typdeklarationen. Der Compiler übersetzt jeden Typ einer Compilationseinheit in eine eigene *.class*-Datei. Ein Paket ist letztendlich eine Sammlung aus Compilationseinheiten. In der Regel ist die Compilationseinheit eine Quellcodedatei; die Codezeilen könnten grundsätzlich auch aus einer Datenbank kommen oder zur Laufzeit generiert werden.

3.6.10 Statischer Import *

Die `import`-Deklaration informiert den Compiler über die Pakete, sodass ein Typ nicht mehr voll qualifiziert werden muss, wenn er im `import`-Teil explizit aufgeführt wird oder wenn das Paket des Typs über `*` genannt ist.

Falls eine Klasse statische Methoden oder Konstanten vorschreibt, werden ihre Eigenschaften immer über den Typnamen angesprochen. Java bietet mit dem *statischen Import* die Möglichkeit, die statischen Methoden oder Variablen ohne vorangestellten Typnamen sofort zu nutzen. Während also das normale `import` dem Compiler Typen benennt, macht ein statisches `import` dem Compiler Klasseneigenschaften bekannt, geht also eine Ebene tiefer.

Beispiel

Importiere die statische Konstante `PI` aus der Klasse `Math`, um sie direkt ohne Klassennamen verwenden zu können:

```
import static java.lang.Math.PI;
```

Normalerweise müsste es beim Zugriff auf die Konstante `Math.PI` heißen. Durch den statischen Import kann der Klassenname entfallen, und es heißt einfach z. B. einfach `area = PI * radius * radius`; Das verkürzt mathematische Ausdrücke und erhöht oft die Lesbarkeit, insbesondere bei Formeln.



Binden wir in einem Beispiel mehrere statische Eigenschaften mit einem statischen `import` ein:

Listing 3.11 `src/main/java/com/tutego/insel/ooop/StaticImport.java`

```
import static java.lang.System.out;
import static javax.swing.JOptionPane.showInputDialog;
import static java.lang.Integer.parseInt;
import static java.lang.Math.max;
import static java.lang.Math.min;

static void main() {
    int i = parseInt( showInputDialog( "First number" ) );
    int j = parseInt( showInputDialog( "Second number" ) );
```

```
out.printf( "%d is greater than or equal to %d.%n",
            max(i, j), min(i, j) );
}
```

Mehrere Typen statisch importieren

Der statische Import

```
import static java.lang.Math.max;
import static java.lang.Math.min;
```

bindet die statische `max(...)/min(...)`-Methode ein. Besteht Bedarf an weiteren statischen Methoden, gibt es neben der individuellen Aufzählung eine Wildcard-Variante:

```
import static java.lang.Math.*;
```



Best Practice

Auch wenn Java diese Möglichkeit bietet, sollte der Einsatz maßvoll erfolgen. Die Möglichkeit der statischen Importe ist nützlich, wenn Klassen Konstanten nutzen wollen. Allerdings besteht auch die Gefahr, dass durch den fehlenden Typnamen nicht mehr sichtbar ist, woher die Eigenschaft eigentlich kommt und welche Abhängigkeit sich damit aufbaut. Auch gibt es Probleme mit gleichlautenden Methoden: Eine Methode aus der eigenen Klasse überdeckt statisch importierte Methoden. Wenn also später in der eigenen Klasse – oder Oberklasse – eine Methode aufgenommen wird, die die gleiche Signatur hat wie eine statisch importierte Methode, wird das zu keinem Compilerfehler führen, sondern die Semantik wird sich ändern, weil jetzt die neue eigene Methode verwendet wird und nicht mehr die statisch importierte.

3.7 Mit Referenzen arbeiten, Vielfalt, Identität, Gleichwertigkeit

In Java gibt es mit `null` eine sehr spezielle Referenz, die Auslöser vieler Probleme ist. Doch ohne sie geht es nicht, und warum das so ist, wird der folgende Abschnitt zeigen. Anschließend wollen wir sehen, wie Objektvergleiche funktionieren und was der Unterschied zwischen Identität und Gleichwertigkeit ist.

3.7.1 null-Referenz und die Frage der Philosophie

In Java gibt es drei spezielle Referenzen: `null`, `this` und `super`. (Wir verschieben die Beschreibung von `this` und `super` auf Kapitel 6, »Eigene Klassen schreiben«.) Das spezielle Literal `null` lässt sich zur Initialisierung von Referenzvariablen verwenden. Die `null`-Referenz ist typenlos, kann also jeder Referenzvariablen zugewiesen und jeder Methode übergeben werden, die ein Objekt erwartet.⁸

⁸ `null` verhält sich also so, als ob es ein Untertyp jedes anderen Typs wäre.

**Beispiel**

Deklaration und Initialisierung zweier Objektvariablen mit `null`:

```
Point p = null;
String s = null;
IO.println( p ); // null
```

Die Konsolenausgabe über die letzte Zeile liefert kurz »null«. Wir haben hier die String-Repräsentation vom `null`-Typ vor uns.

Da `null` typenlos ist und es nur ein `null` gibt, kann `null` zu jedem Typ typangepasst werden, und so ergibt zum Beispiel `(String) null == null && (Point) null == null` das Ergebnis `true`. Das Literal `null` ist ausschließlich für Referenzen vorgesehen und kann in keinen primitiven Typ wie die Ganzzahl `0` umgewandelt werden.⁹

Mit `null` lässt sich eine ganze Menge machen. Der Haupteinsatzzweck sieht vor, damit uninitialisierte Referenzvariablen zu kennzeichnen, also auszudrücken, dass eine Referenzvariable auf kein Objekt verweist. In Listen oder Bäumen kennzeichnet `null` zum Beispiel das Fehlen eines gültigen Nachfolgers oder bei einem grafischen Dialog, dass der Benutzer den Dialog abgebrochen hat; `null` ist dann ein gültiger Indikator und kein Fehlerfall.

**Hinweis**

Bei einer mit `null` initialisierten lokalen Variablen funktioniert die Abkürzung mit `var` nicht; es gibt einen Compilerfehler:

```
var text = null; // ☠ Cannot infer type: variable initializer is 'null'
```

Auf `null` geht nix, nur die `NullPointerException`

Da sich hinter `null` kein Objekt verbirgt, ist es auch nicht möglich, eine Methode aufzurufen oder von `null` eine Objektvariable zu erfragen. Der Compiler kennt zwar den Typ jedes Ausdrucks, aber erst die Laufzeitumgebung (JVM) weiß, was referenziert wird. Bei dem Versuch, über die `null`-Referenz auf eine Eigenschaft eines Objekts zuzugreifen, löst eine JVM eine `NullPointerException`¹⁰ aus:

⁹ Hier unterscheiden sich C(++) und Java.

¹⁰ Der Name zeigt das Überbleibsel von Zeigern. Zwar haben wir es in Java nicht mit Zeigern zu tun, sondern mit Referenzen, doch heißt es `NullPointerException` und nicht `NullReferenceException`. Das erinnert daran, dass eine Referenz ein Objekt identifiziert und eine Referenz auf ein Objekt ein Pointer ist. Das .NET Framework ist hier konsequenter und nennt die Ausnahme `NullReferenceException`.

Listing 3.12 src/main/java/com/tutego/insel/oop/NullPointer.java

```
static void main() {                                // 1
    java.awt.Point p = null;                        // 2
    String        s = null;                        // 3
    p.setLocation( 1, 2 );                          // 4
    s.length();                                     // 5
}                                                    // 6
```

Wir beobachten eine `NullPointerException` zur Laufzeit, denn das Programm bricht bei `p.setLocation(...)` mit folgender Ausgabe ab:

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.awt.Point.setLocation(int, int)" because "p" is null
    at NullPointer.main(NullPointer.java:4)
```

Die Laufzeitumgebung teilt uns in der Fehlermeldung mit, dass sich der Fehler, die `NullPointerException`, in Zeile 4 befindet. Um den Fehler zu korrigieren, müssen wir entweder die Variablen initialisieren, das heißt, ein Objekt zuweisen wie in

```
p = new java.awt.Point();
s = "";
```

oder vor dem Zugriff auf die Eigenschaften einen Test durchführen, ob Objektvariablen auf etwas zeigen oder `null` sind, und in Abhängigkeit vom Ausgang des Tests den Zugriff auf die Eigenschaft zulassen oder nicht.

3.7.2 Alles auf null? Referenzen testen

Mit dem Vergleichsoperator `==` oder dem Test auf Ungleichheit mit `!=` lässt sich leicht herausfinden, ob eine Referenzvariable wirklich ein Objekt referenziert oder nicht:

```
if ( object == null )
    // Variable referenziert nichts, ist aber gültig mit null initialisiert
else
    // Variable referenziert ein Objekt
```

null-Test und Kurzschluss-Operatoren

Wir wollen an dieser Stelle noch einmal auf die üblichen logischen Kurzschluss-Operatoren und den logischen, nicht kurzschließenden Operator zu sprechen kommen. Erstere werten Operanden nur so lange von links nach rechts aus, bis das Ergebnis der Operation feststeht. Auf den ersten Blick scheint es nicht viel auszumachen, ob alle Teilausdrücke ausgewertet werden oder nicht. In einigen Ausdrücken ist dies aber wichtig, wie das folgende Beispiel für

die Variable `s` vom Typ `String` zeigt; das Programm soll die `String`-Länge ausgeben, wenn ein `String` eingegeben wurde:

Listing 3.13 `src/main/java/NullCheckForStringLength.java`

```
String s = javax.swing.JOptionPane.showInputDialog( "Input a string" );
if ( s != null && ! s.isEmpty() )
    IO.println( "Length of string: " + s.length() );
else
    IO.println( "Dialog cancelled or no input given" );
```

Die Rückgabe von `showInputDialog(...)` ist `null`, wenn der Benutzer den Dialog abbricht. Das soll unser Programm berücksichtigen. Daher testet die `if`-Bedingung, ob `s` überhaupt auf ein Objekt verweist, und wenn ja, zusätzlich, ob der `String` nicht leer ist. Dann folgt eine Ausgabe.

Diese Schreibweise tritt häufig auf, und der Und-Operator zur Verknüpfung muss ein Kurzschluss-Operator sein, da es in diesem Fall ausdrücklich darauf ankommt, dass die Länge nur dann bestimmt wird, wenn die Variable `s` überhaupt auf ein `String`-Objekt verweist und nicht `null` ist. Andernfalls bekämen wir bei `s.isEmpty()` eine `NullPointerException`, wenn jeder Teilausdruck ausgewertet würde und `s` gleich `null` wäre.

Das Glück der anderen: Null Coalescing Operator *

Wenn `null`-Referenzen erlaubt sind, muss im Code häufig sichergestellt werden, dass für den Fall eines `null`-Werts eine sinnvolle Alternative verwendet wird, da andernfalls eine `null`-Referenzierung zu einer `NullPointerException` führt. Aus diesem Grund gibt es oft Code, der eine Fallback-Logik für `null`-Werte implementiert. Ein typisches Muster verwendet dabei den Bedingungsoperator in der Form `o != null ? o : non_null_o`. Viele Programmiersprachen wie JavaScript, Kotlin, Objective-C, PHP oder Swift bieten dafür eine kürzere Syntax: den *Null Coalescing Operator*; *coalescing* bedeutet auf Deutsch etwa »verschmelzend«. Dieser Operator erlaubt es, einen Standardwert anzugeben, falls ein Ausdruck `null` ist. In C# sieht das Beispiel so aus: `o ?? non_null_o`. Elegant ist das bei verketteten Tests der Art `o ?? p ?? q ?? r`, wo es dann sinngemäß heißt: »Liefere die erste Referenz ungleich `null`.« Java bietet keinen solchen Operator.



3.7.3 Zuweisungen bei Referenzen

Eine Referenz erlaubt den Zugriff auf das referenzierte Objekt, und eine Referenzvariable speichert eine Referenz. Es kann durchaus mehrere Referenzvariablen geben, die die gleiche Referenz speichern. Das wäre so, als ob ein Objekt unter verschiedenen Namen angesprochen wird – so wie eine Person von den Mitarbeitern als »Chefin« angesprochen wird, aber von ihrem Mann als »Schnuckiputzi«. Dies nennt sich auch *Alias*.



[zB]

Beispiel

Ein Punkt-Objekt wollen wir unter einem alternativen Variablennamen ansprechen:

```
Point p = new Point();
Point q = p;
```

Ein Punkt-Objekt wird erzeugt und mit der Variablen `p` referenziert. Die zweite Zeile speichert nun dieselbe Referenz in der Variablen `q`. Danach verweisen `p` und `q` auf dasselbe Objekt. Zum besseren Verständnis: Wichtig ist, wie oft es `new` gibt, denn das sagt aus, wie viele Objekte die JVM bildet. Und bei den zwei Zeilen gibt es nur ein `new`, also auch nur einen Punkt.

Verweisen zwei Objektvariablen auf dasselbe Objekt, hat das natürlich zur Konsequenz, dass über zwei Wege Objektzustände ausgelesen und modifiziert werden können. Heißt die gleiche Person in der Firma »Chefin« und zu Hause »Schnuckiputzi«, wird der Mann sich freuen, wenn die Frau in der Firma keinen Stress hat.

Wir können das Beispiel auch gut bei Punkt-Objekten nachverfolgen. Zeigen `p` und `q` auf dasselbe Punkt-Objekt, können Änderungen über die Variable `p` auch über die Variable `q` beobachtet werden:

Listing 3.14 `src/main/java/com/tutego/insel/oop/ItsTheSame.java`

```
import java.awt.*;

static void main() {
    Point p = new Point();
    Point q = p;
    p.x = 10;
    IO.println( q.x ); // 10
}
```

```

    q.y = 5;
    IO.println( p.y ); // 5
}

```

3.7.4 Methoden mit Referenztypen als Parameter

Dass sich dasselbe Objekt unter zwei Namen (über zwei verschiedene Variablen) ansprechen lässt, können wir gut bei Methoden beobachten. Eine Methode, die über den Parameter eine Objektreferenz erhält, kann auf das übergebene Objekt zugreifen. Das bedeutet, die Methode kann dieses Objekt mit den angebotenen Methoden ändern oder auf die Objektvariablen zugreifen.

Im folgenden Beispiel deklarieren wir zwei Methoden. Die erste Methode, `initializePosition(Point)`, soll einen übergebenen Punkt mit Zufallskordinaten initialisieren. Übergeben werden der Methode in `main(...)` später zwei `Point`-Objekte: einmal für einen Spieler und einmal für eine Schlange. Die zweite Methode, `printScreen(Point, Point)`, gibt das Spielfeld auf dem Bildschirm aus und gibt dann, wenn die Koordinate einen Spieler trifft, ein `&` aus und bei der Schlange ein `S`. Falls Spieler und Schlange zufälligerweise zusammentreffen, »gewinnt« die Schlange.

Listing 3.15 `src/main/java/DrawPlayerAndSnake.java`

```

import java.awt.Point;

static void initializePosition( Point p ) {
    int randomX = (int)(Math.random() * 40); // 0 <= x < 40
    int randomY = (int)(Math.random() * 10); // 0 <= y < 10
    p.setLocation( randomX, randomY );
}

static void printScreen( Point playerPosition,
                        Point snakePosition ) {
    for ( int y = 0; y < 10; y++ ) {
        for ( int x = 0; x < 40; x++ ) {
            if ( snakePosition.distanceSq( x, y ) == 0 )
                IO.print( 'S' );
            else if ( playerPosition.distanceSq( x, y ) == 0 )
                IO.print( '&' );
            else IO.print( '.' );
        }
        IO.println();
    }
}

```

```
static void main() {  
    Point playerPosition = new Point();  
    Point snakePosition = new Point();  
    IO.println( playerPosition );  
    IO.println( snakePosition );  
    initializePosition( playerPosition );  
    initializePosition( snakePosition );  
    IO.println( playerPosition );  
    IO.println( snakePosition );  
    printScreen( playerPosition, snakePosition );  
}
```

Die Ausgabe kann so aussehen:

```
java.awt.Point[x=0,y=0]  
java.awt.Point[x=0,y=0]  
java.awt.Point[x=38,y=1]  
java.awt.Point[x=19,y=8]  
.....  
.....&.  
.....  
.....  
.....  
.....  
.....  
.....S.....  
.....
```

In dem Moment, in dem `main(...)` die statische Methode `initializePosition(Point)` aufruft, gibt es sozusagen zwei Namen für das `Point`-Objekt: `playerPosition` und `p`. Allerdings ist das nur innerhalb der virtuellen Maschine so, denn `initializePosition(Point)` kennt das Objekt nur unter `p`, aber kennt die Variable `playerPosition` nicht. Bei `main(...)` ist es umgekehrt: Nur der Variablenname `playerPosition` ist in `main(...)` bekannt, er hat aber vom Namen `p` keine Ahnung. Die `Point`-Methode `distanceSq(int, int)` liefert den quadrierten Abstand vom aktuellen Punkt zu den übergebenen Koordinaten.



Hinweis

Der Name einer Parametervariablen darf durchaus mit dem Namen der Argumentvariablen übereinstimmen, was die Semantik nicht verändert. Die Namensräume sind völlig getrennt, und Missverständnisse gibt es nicht, da beide – die aufrufende Methode und die aufgerufene Methode – komplett getrennte Variablen haben.

Wertübergabe und Referenzübergabe per Call by Value

Primitive Variablen werden immer per Wert kopiert (*Call by Value*). Das Gleiche gilt für Referenzen, die als eine Art Zeiger zu verstehen sind, und das sind im Prinzip nur Ganzzahlen. Daher hat auch die folgende statische Methode keine Nebenwirkungen:

Listing 3.16 src/main/java/com/tutego/insel/oop/JavalsAlwaysCallByValue.java

```
import java.awt.Point;

static void clear( Point p ) {
    IO.println( p ); // java.awt.Point[x=10,y=20]
    p = new Point();
    IO.println( p ); // java.awt.Point[x=0,y=0]
}

static void main() {
    Point p = new Point( 10, 20 );
    clear( p );
    IO.println( p ); // java.awt.Point[x=10,y=20]
}
```

Nach der Zuweisung `p = new Point()` in der `clear(Point)`-Methode referenziert die Parametervariable `p` ein anderes Punkt-Objekt, und der an die Methode übergebene Verweis geht damit verloren. Diese Änderung wird nach außen hin natürlich nicht sichtbar, denn die Parametervariable `p` von `clear(...)` ist nur ein temporärer alternativer Name für das `p` aus `main`; eine Neu-zuweisung an das `clear-p` ändert nicht den Verweis vom `main-p`. Das bedeutet, dass der Aufrufer von `clear(...)` – und das ist `main(...)` – kein neues Objekt unter sich hat. Wer den Punkt mit null initialisieren möchte, muss auf die Zustände des übergebenen Objekts direkt zugreifen, etwa so:

```
static void clear( Point p ) {
    p.x = p.y = 0;
}
```

Call by Reference gibt es in Java nicht – ein Blick auf C und C++ *

In C++ gibt es eine weitere Argumentübergabe, die sich *Call by Reference* nennt. Eine `swap(...)`-Funktion ist ein gutes Beispiel für die Nützlichkeit von Call by Reference:

```
void swap( int& a, int& b ) { int tmp = a; a = b; b = tmp; }
```

Zeiger und Referenzen sind in C++ etwas anderes, was Spracheinsteiger leicht irritiert. Denn in C++ und auch in C hätte eine vergleichbare `swap(...)`-Funktion auch mit Zeigern implementiert werden können:



```
void swap( int *a, int *b ) { int tmp = *a; *a = *b; *b = tmp; }
```

Die Implementierung gibt in C(++) einen Verweis auf das Argument.

Final deklarierte Referenzparameter und das fehlende const

Wir haben gesehen, dass finale Variablen nicht erneut beschrieben werden dürfen. Final können lokale Variablen, Parametervariablen, Objektvariablen oder Klassenvariablen sein. In jedem Fall sind neue Zuweisungen tabu. Dabei ist es egal, ob die Parametervariable vom primitiven Typ oder vom Referenztyp ist. Bei einer Methodendeklaration der folgenden Art wäre also eine Zuweisung an `p` und auch an `value` verboten:

```
public void clear( final Point p, final int value )
```

Ist die Parametervariable nicht `final` und ein Referenztyp, so würden wir mit einer Zuweisung den Verweis auf das ursprüngliche Objekt verlieren, und das wäre wenig sinnvoll, wie wir im vorangehenden Beispiel gesehen haben. `final` deklarierte Parametervariablen machen im Programmcode deutlich, dass eine Änderung der Referenzvariablen unsinnig ist, und der Compiler verbietet eine Zuweisung. Im Fall unserer `clear(...)`-Methode wäre die Initialisierung direkt als Compilerfehler aufgefallen:

```
static void clear( final Point p ) {
    p = new Point();    // ☠ Cannot assign a value to final variable 'p'
}
```

Halten wir fest: Ist ein Parameter mit `final` deklariert, sind keine Zuweisungen möglich. `final` verbietet aber keine Änderungen an Objekten – und so könnte `final` im Sinne der Übersetzung als »endgültig« verstanden werden. Mit der Referenz des Objekts können wir sehr wohl den Zustand verändern, so wie wir es auch im letzten Beispielprogramm taten.

`final` erfüllt demnach nicht die Aufgabe, schreibende Objektzugriffe zu verhindern. Eine Methode mit übergebenen Referenzen kann also Objekte verändern, wenn es etwa `set*(...)`-Methoden oder Variablen gibt, auf die zugegriffen werden kann. Die Dokumentation muss also immer ausdrücklich beschreiben, wann die Methode den Zustand eines Objekts modifiziert.

Sprachenvergleich *

In C++ gibt es für Parameter den Zusatz `const`, an dem der Compiler erkennen kann, dass Objektzustände nicht verändert werden sollen. Ein Programm nennt sich *const-korrekt*, wenn es niemals ein konstantes Objekt verändert. Dieses `const` ist in C++ eine Erweiterung des Objekttyps, die es in Java nicht gibt. Zwar haben die Java-Entwickler das Schlüsselwort `const` reserviert, doch genutzt wird es bisher nicht.

3.7.5 Identität von Objekten

Die Vergleichsoperatoren `==` und `!=` sind für alle Datentypen so definiert, dass sie die vollständige Übereinstimmung zweier Werte testen. Bei primitiven Datentypen ist das einfach einzusehen und bei Referenztypen im Prinzip genauso (zur Erinnerung: Referenzen lassen sich als Pointer verstehen, was Ganzzahlen sind). Der Operator `==` testet bei Referenzen, ob sie übereinstimmen, also auf dasselbe Objekt verweisen. Der Operator `!=` testet das Gegenteil, also ob sie nicht übereinstimmen, die Referenzen somit ungleich sind. Demnach sagt der Test etwas über die Identität der referenzierten Objekte aus, aber nichts darüber, ob zwei verschiedene Objekte möglicherweise den gleichen Inhalt haben. Der Inhalt der Objekte spielt bei `==` und `!=` keine Rolle.

Beispiel

Zwei Objekte mit drei unterschiedlichen Punktvariablen `p`, `q`, `r` und die Bedeutung von `==`:

```
Point p = new Point( 10, 10 );
Point q = p;
Point r = new Point( 10, 10 );
IO.println( p == q ); // true, da p und q dasselbe Objekt referenzieren
IO.println( p == r ); // false, da p und r zwei verschiedene Punkt-
                       // Objekte referenzieren, die zufällig dieselben
                       // Koordinaten haben
```

Da `p` und `q` auf dasselbe Objekt verweisen, ergibt der Vergleich `true`. `p` und `r` referenzieren unterschiedliche Objekte, die aber zufälligerweise den gleichen Inhalt haben. Doch woher soll der Compiler wissen, wann zwei Punkt-Objekte inhaltlich gleich sind? Weil sich ein Punkt durch die Objektvariablen `x` und `y` auszeichnet? Die Laufzeitumgebung könnte voreilig die Belegung jeder Objektvariablen vergleichen, doch das entspricht nicht immer einem korrekten Vergleich, so wie wir ihn uns wünschen. Ein Punkt-Objekt könnte etwa zusätzlich die Anzahl der Zugriffe zählen oder einen Zeitpunkt für den letzten Zugriff speichern, doch für einen Vergleich der Lage zweier Punkte sind diese Informationen dann irrelevant.

3.7.6 Gleichwertigkeit und die Methode `equals(...)`

Die allgemeingültige Lösung besteht darin, die Klasse festlegen zu lassen, wann Objekte gleich(wertig) sind. Dazu kann jede Klasse eine Methode `equals(...)` implementieren, und mit ihrer Hilfe kann sich jedes Exemplar dieser Klasse mit beliebigen anderen Objekten vergleichen. Die Klassen entscheiden immer nach Anwendungsfall, welche Objektvariablen sie für einen Gleichheitstest heranziehen, und `equals(...)` liefert `true`, wenn die gewünschten Zustände (Objektvariablen) übereinstimmen.



**Beispiel**

Zwei nicht identische, inhaltlich gleiche Punkt-Objekte werden mit `==` und `equals(...)` verglichen:

```
Point p = new Point( 10, 10 );
Point q = new Point( 10, 10 );
IO.println( p == q );      // false
IO.println( p.equals(q) ); // true, da symmetrisch auch q.equals(p)
```

Nur `equals(...)` testet in diesem Fall die inhaltliche Gleichwertigkeit.

Bei den unterschiedlichen Bedeutungen müssen wir demnach die Begriffe *Identität* und *Gleichwertigkeit* (auch *Gleichheit*) von Objekten sorgfältig unterscheiden. Tabelle 3.4 zeigt noch einmal eine Zusammenfassung.

	Getestet mit	Implementierung
Identität der Referenzen	<code>==</code> bzw. <code>!=</code>	nichts zu tun
Gleichwertigkeit der Zustände	<code>equals(...)</code> bzw. <code>! equals(...)</code>	abhängig von der Klasse

Tabelle 3.4 Identität und Gleichwertigkeit von Objekten

`equals(...)`-Implementierung von `Point` *

Die Klasse `Point` deklariert `equals(...)`, wie die API-Dokumentation zeigt. Werfen wir einen Blick auf die Implementierung, um eine Vorstellung von der Arbeitsweise zu bekommen:

Listing 3.17 `java/awt/Point.java`

```
public class Point ... {

    public int x;
    public int y;
    ...
    public boolean equals( Object obj ) {
        ...
        Point pt = (Point) obj;
        return (x == pt.x) && (y == pt.y); // (*)
        ...
    }
}
```

Obwohl bei diesem Beispiel für uns einiges neu ist, erkennen wir den Vergleich in der Zeile (*). Hier vergleicht das `Point`-Objekt seine eigenen Objektvariablen mit den Objektvariablen des `Punkt`-Objekts, das als Argument an `equals(...)` übergeben wurde.

Es gibt immer ein equals(...) – die Oberklasse Object und ihr equals(...)*

Glücklicherweise ist es nicht nötig, lange darüber nachzudenken, ob eine Klasse eine equals(...) -Methode anbieten soll oder nicht. Jede Klasse besitzt sie, da die universelle Oberklasse Object sie vererbt. Wir greifen hier auf Kapitel 7, »Objektorientierte Beziehungsfragen«, vor; dieser Abschnitt kann aber übersprungen werden. Wenn eine Klasse also keine eigene equals(...) -Methode angibt, dann erbt sie eine Implementierung aus der Klasse Object. Diese Klasse sieht wie folgt aus:

Listing 3.18 java/lang/Object.java

```
public class Object {
    public boolean equals( Object obj ) {
        return ( this == obj );
    }
    ...
}
```

Wir erkennen, dass hier die Gleichwertigkeit auf die Identität der Referenzen abgebildet wird. Ein inhaltlicher Vergleich findet nicht statt. Das ist das Einzige, was die vorgegebene Implementierung machen kann, denn sind die Referenzen identisch, sind die Objekte logischerweise auch gleich. Nur über Zustände »weiß« die Basisklasse Object nichts.

Sprachvergleich

Es gibt Programmiersprachen, die für Identitätsvergleich und Gleichwertigkeitstest eigene Operatoren anbieten. Was bei Java == und equals(...) sind, sind bei Python is und ==, bei Swift === und ==.

**3.7.7 Ausblick: Value Types – Objekte ohne Identität**

In Java ist die Identität eines Objekts ein zentrales Konzept. Jedes mit new erzeugte Objekt besitzt eine eindeutige Identität, unabhängig von seinen Attributwerten. Selbst wenn zwei Objekte exakt den gleichen Zustand aufweisen, bleiben sie dennoch getrennte Instanzen, da ihre Referenzen unterschiedlich sind. Deshalb ist eine sauber implementierte equals(...) -Methode so entscheidend, denn der ==-Operator prüft lediglich die Identität, nicht die Gleichheit der Werte.

Seit vielen Jahren arbeitet das Java-Team an *Value Types*, einer neuen Kategorie von Datentypen, die zwischen primitiven Typen und klassischen Referenztypen angesiedelt sind. Der entscheidende Unterschied: Value Types besitzen keine Identität und werden ausschließlich durch ihre Werte definiert. Das bedeutet, dass zwei Value-Objekte mit gleichem Zustand tatsächlich als identisch gelten. Dadurch ändert sich auch die Semantik des ==-Operators: An-

statt Referenzen zu vergleichen, prüft er bei Value Types, ob deren Werte übereinstimmen, ähnlich wie bei primitiven Datentypen. Damit das Konzept funktioniert, müssen Value Types immutable sein.

Um dieses neue Paradigma zu unterstützen, werden Value Types mit einer speziellen Syntax eingeführt, die sich von herkömmlichen Klassendeklarationen unterscheidet. Die Einführung von Value Types bringt Performancevorteile und ermöglicht eine elegante Modellierung wertbasierter Datentypen etwa für Zeit, Geld oder Vektoren – Fälle, in denen die Identität eines Objekts keine Rolle spielt, sondern nur seine Werte.

Interessenten können unter <https://openjdk.org/jeps/401> mehr zu diesem geplanten Sprachfeature erfahren.

3.8 Der Zusammenhang von new, Heap und Garbage-Collector

Die JVM-Spezifikation sieht für Daten verschiedene Speicherbereiche (engl. *runtime data areas*) vor.¹¹ Im Mittelpunkt stehen der *Heap-Speicher* und der *Stack-Speicher* (Stapelspeicher). Java nutzt den Stack-Speicher, um die Reihenfolge von Methodenaufrufen zu verwalten sowie übergebene Argumente bei Methodenaufrufen und lokale Variablen zu speichern. Bei endlosen rekursiven Methodenaufrufen, typischerweise durch fehlerhafte Rekursion, kann die maximale Stack-Größe überschritten werden, was zu einer `java.lang.StackOverflowError`-Exception führt. Da jeder Thread einen eigenen JVM-Stack hat, führt dies zum Ende des betroffenen Threads, während andere Threads davon unbeeinträchtigt weiterlaufen.

3.8.1 Heap-Speicher

Wenn das Laufzeitsystem die Anfrage erhält, ein Objekt mit `new` zu erzeugen, reserviert es ausreichend Speicher, um alle Eigenschaften des Objekts sowie Verwaltungsinformationen (Metadaten wie der Objekttyp oder Informationen für den Garbage-Collector) unterzubringen. Zum Beispiel speichert ein `Point`-Objekt die Koordinaten in zwei `int`-Werten, was mindestens 2 mal 4 Byte Speicher erfordert. Dieser Speicherplatz wird aus dem Heap bezogen.



Hinweis

Es gibt in Java nur wenige Sonderfälle, in denen neue Objekte nicht über `new` angelegt werden. So erzeugt die auf nativem Code basierende Methode `newInstance()` vom `Constructor`-Objekt ein neues Objekt. Auch `clone()` kann ein neues Objekt als Kopie eines anderen Objekts

11 § 2.5 der JVM-Spezifikation, <https://docs.oracle.com/javase/specs/jvms/se25/html/jvms-2.html#jvms-2.5>.

erzeugen. Bei der String-Konkatenation mit `+` ist für uns zwar kein `new` zu sehen, doch der Compiler wird Anweisungen bauen, um das neue `String`-Objekt anzulegen.

Der Heap wächst von einer Startgröße bis zu einer maximal erlaubten Größe, um sicherzustellen, dass ein Java-Programm nicht unbeschränkt viel Speicher vom Betriebssystem abrufen, was die Maschine möglicherweise in den Ruin treibt. In der HotSpot-JVM beträgt die Startgröße des Heaps $\frac{1}{64}$ des Hauptspeichers und wächst dann bis zu einer maximalen Größe von $\frac{1}{4}$ des Hauptspeichers.¹²

3.8.2 Automatische Speicherbereinigung/Garbage-Collector (GC) – es ist dann mal weg

Nehmen wir folgendes Szenario an:

```
java.awt.Point candyStoreLocation;
candyStoreLocation = new java.awt.Point( 50, 9 );
candyStoreLocation = new java.awt.Point( 51, 7 );
```

Wir deklarieren eine `Point`-Variable, bauen ein Exemplar auf und belegen die Variablen. Dann bauen wir ein neues `Point`-Objekt auf und überschreiben die Variable. Doch was ist mit dem ersten Punkt?

Wird das Objekt nicht mehr vom Programm referenziert, so bemerkt dies die automatische Speicherbereinigung alias der Garbage-Collector (GC) und gibt den reservierten Speicher wieder frei.¹³ Die automatische Speicherbereinigung testet dazu regelmäßig, ob die Objekte auf dem Heap noch benötigt werden. Werden sie nicht benötigt, löscht der Objektjäger sie. Es weht also immer ein Hauch von Friedhof über dem Heap, und nachdem die letzte Referenz vom Objekt genommen wurde, ist es auch schon tot. Es gibt verschiedene GC-Algorithmen, und jeder Hersteller einer JVM hat eigene Verfahren.

3.8.3 OutOfMemoryError

Ist das System nicht in der Lage, genügend Speicher für ein neues Objekt bereitzustellen, versucht die automatische Speicherbereinigung in einer letzten Rettungsaktion, alles Ungebrauchte wegzuräumen. Ist dann immer noch nicht ausreichend Speicher frei, generiert die Laufzeitumgebung einen `OutOfMemoryError` und beendet das gesamte Programm.¹⁴

¹² <https://docs.oracle.com/en/java/javase/25/gctuning/ergonomics.html>

¹³ Mit dem gesetzten `java`-Schalter `-verbose:gc` gibt es immer Konsolenausgaben, wenn der GC nicht mehr referenzierte Objekte erkennt und wegräumt.

¹⁴ Diese besondere Ausnahme kann aber auch abgefangen werden. Das ist für den Serverbetrieb wichtig, denn wenn ein Puffer zum Beispiel nicht erzeugt werden kann, soll nicht gleich die ganze JVM stoppen.

**Beispiel**

Wir provozieren einen `OutOfMemoryError` mit folgender Zeile:

```
for ( String s = " "; ; s += s + s ) ;
```

Dieses Programm verdoppelt die Größe des Strings `s` in jeder Iteration, bis der Heap voll ist.

3.9 Zum Weiterlesen

In diesem Kapitel wurde das Thema Objektorientierung recht schnell eingeführt, was nicht bedeuten soll, dass OOP einfach ist. Der Weg zu gutem Design ist steinig und führt nur über viele Java-Projekte. Hilfreich sind das Lesen von fremden Programmen und die Beschäftigung mit Entwurfsmustern. Leserinnen und Leser sollten sich mit UML vertraut machen, um Designideen skizzieren zu können. Einen interessanten Ansatz verfolgt PlantUML (<https://plantuml.com/>) mit einer Textsyntax, die das Werkzeug in Grafiken konvertiert.

Kapitel 16

Die Klassenbibliothek

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

16.1 Die Java-Klassenphilosophie

Eine Programmiersprache besteht nicht nur aus einer Grammatik, sondern – wie im Fall von Java – auch aus einer umfangreichen Standardbibliothek. C++ gilt zwar als plattformunabhängig, da der Sprachstandard (ISO C++) klar definiert ist und es Compiler für nahezu jede Plattform gibt, in der Praxis ist der Anteil der standardisierten Bibliothek jedoch klein im Vergleich zu dem, was typische Anwendungen benötigen. Für grafische Oberflächen, Netzwerk- oder Datenbankzugriffe werden häufig spezielle Bibliotheken eingesetzt, die auf betriebssystemspezifischen Funktionen basieren. Während sich plattformneutrale Algorithmen in C++ problemlos übertragen lassen, erfordern Ein- und Ausgabe oder GUI-Programmierung oft Anpassungen. Die Java-Standardbibliothek dagegen abstrahiert weitgehend von plattform-spezifischen Details und stellt zentrale Funktionen in konsistenten, objektorientierten Klassen und Paketen bereit – insbesondere für Datenstrukturen, Ein- und Ausgabe, Grafik- und Netzwerkprogrammierung.

16.1.1 Modul, Paket, Typ

An oberster Stelle der Java-Bibliothek stehen Module. Sie wiederum bestehen aus Paketen, die wiederum die Typen enthalten.



Module der Java SE

Die Java-Plattform ist modular aufgebaut. Anstelle eines monolithischen JDK besteht die Java SE aus vielen einzelnen Modulen. Jedes Modul enthält klar abgegrenzte Funktionalität und kann gezielt verwendet werden. Das zentrale Modul ist `java.base` und enthält Kernklassen wie `Object` und `String` usw. Es ist das einzige Modul, das selbst keine Abhängigkeit zu anderen Modulen enthält. Jedes andere Modul jedoch bezieht sich mindestens auf `java.base`. Die Javadoc stellt das grafisch schön dar (siehe Abbildung 16.1).

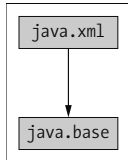


Abbildung 16.1 Das Modul »java.xml« hat eine Abhängigkeit zum »java.base«-Modul.

Stellenweise gibt es mehr Abhängigkeiten, etwa beim Modul `java.desktop`, wie Abbildung 16.2 demonstriert:

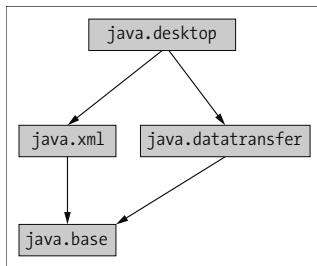


Abbildung 16.2 Abhängigkeiten des Moduls »java.desktop«

Das Modul java.se

Ein besonderes Modul ist `java.se`. Es deklariert selbst keine eigenen Pakete oder Typen, sondern fasst lediglich andere Module zusammen. Der Name für eine solche Konstruktion ist *Aggregator-Modul*. Das `java.se`-Modul definiert auf diese Weise die API für die Java SE-Plattform (siehe Abbildung 16.3).

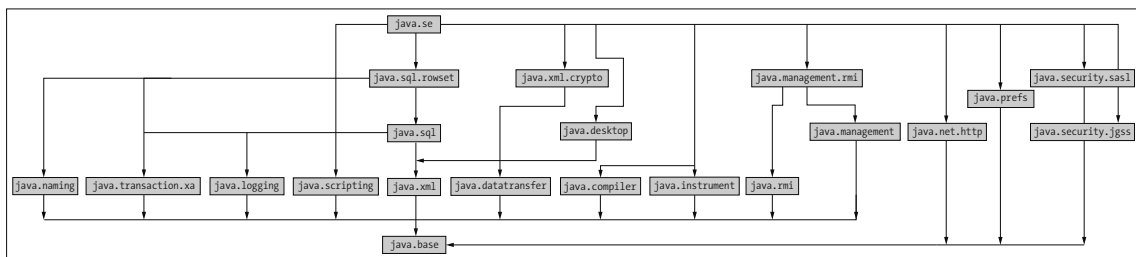


Abbildung 16.3 Abhängigkeiten des Moduls »java.se«

Die *Java 25 Core Java SE API* besteht aus vielen Modulen und Paketen. Eine Kurzbeschreibung befindet sich in Anhang A.

Hinweis

Wir werden im Folgenden bei den Java SE-Typen nicht darauf eingehen, aus welchem Modul sie stammen. Es ist nur dann wichtig zu wissen, in welchem Modul sich ein Typ befindet, wenn kleinere Teilmengen der Java SE gebaut werden.



16.1.2 Übersicht über die Pakete der Standardbibliothek

Die Java-Standardbibliothek besteht aus einer Vielzahl von Paketen, die unterschiedliche Aufgabenbereiche abdecken – von elementaren Sprachfunktionen über Datenstrukturen und Ein-/Ausgabe bis hin zu Netzwerk- und GUI-Programmierung. Ein grundlegendes Verständnis der wichtigsten Pakete erleichtert die Orientierung in der API und unterstützt bei der Auswahl passender Klassen für konkrete Programmieraufgaben.

Die folgende Tabelle gibt einen Überblick über zentrale Pakete und ihre typischen Einsatzbereiche:

Paket	Beschreibung
java.awt	Das Paket AWT (<i>Abstract Windowing Toolkit</i>) bietet Klassen zur Grafikausgabe und zur Nutzung von grafischen Bedienoberflächen.
java.awt.event	Schnittstellen für die verschiedenen Ereignisse unter grafischen Oberflächen.
java.io java.nio	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. Enthält unverzichtbare Klassen wie String-, Thread- oder Wrapper-Klassen.
java.net	Kommunikation über Netzwerke. Bietet Klassen zum Aufbau von Client- und Serversystemen, die sich über TCP bzw. 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 sowie für Teile der Internationalisierung und für Zufallszahlen. Unterpakete kümmern sich um reguläre Ausdrücke und Nebenläufigkeit.

Tabelle 16.1 Wichtige Pakete in der Java SE

Paket	Beschreibung
<code>javax.swing</code>	Swing-Komponenten für grafische Oberflächen. Das Paket besitzt diverse Unterpakete.

Tabelle 16.1 Wichtige Pakete in der Java SE (Forts.)

Eine vollständige Übersicht aller Pakete befindet sich in Anhang A, »Java SE-Module und Paketübersicht«. Für Details empfiehlt sich die Java-API-Dokumentation unter <https://docs.oracle.com/en/java/javase/25/docs/api/index.html>, da sie die zentrale Referenz für alle Typen und Pakete der Plattform darstellt.

Offizielle Schnittstelle (java- und javax-Pakete)

Alles, was die Java-Dokumentation aufführt, gilt als offizieller und erlaubter Zugang zur Bibliothek. Diese Typen sind im Grunde für die Ewigkeit gemacht – mit der beruhigenden Aussicht, dass ein heute geschriebenes Java-Programm theoretisch auch noch in 100 Jahren laufen könnte. Doch wer definiert die API? Im Kern sind es drei Quellen:

- ▶ Das Oracle-Entwicklerteam setzt 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.
- ▶ 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 außerdem Klassen, die unterstützt werden, aber nicht Teil der offiziellen API sind. Dazu zählen etwa diverse Swing-Klassen für das Aussehen der Oberfläche.

Standard Extension API (javax-Pakete)

Einige der Java-Pakete beginnen mit `javax`. Dies sind ursprünglich Erweiterungspakete (*Extensions*), die die Kernklassen ergänzen sollten. Im Laufe der Zeit sind jedoch viele der früher zusätzlich einzubindenden Pakete in die Standarddistribution 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 eingebunden werden muss oder ab welcher Java-Version das Paket Teil der Distribution ist. Echte externe Pakete sind unter anderem:

- ▶ die *Java Communications API* für serielle und parallele Schnittstellen
- ▶ die *Java Telephony API*
- ▶ die Spracheingabe/-ausgabe mit der *Java Speech API*

- ▶ *JavaSpaces* für gemeinsamen Speicher unterschiedlicher Laufzeitumgebungen
- ▶ *JXTA* zum Aufbau von P2P-Netzwerken

Schließlich haben wir es beim Entwickeln mit folgenden Bibliotheken zu tun:

1. mit der offiziellen Java-API
2. mit der API aus JSR-Erweiterungen
3. mit nicht offiziellen Bibliotheken, wie quelloffenen Lösungen, etwa zum Zugriff auf PDF-Dateien oder Bankautomaten

Eine wichtige Rolle spielen ebenso Typen aus dem Paket `jakarta`, das Teil von Enterprise Java und quasi-offiziell ist.

16.2 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, 00:00:00 UTC vergangenen Millisekunden zurück.
- `static long nanoTime()`
Liefert die Zeit vom genauesten Systemzeitgeber. Sie hat keinen Bezugspunkt zu einem Datum.

Die Differenz zweier Zeitwerte kann zur groben Abschätzung der Ausführungszeiten von Programmen dienen.

Tipp

Die von `nanoTime()` gelieferten Werte steigen innerhalb einer JVM-Instanz monoton an. Es gibt jedoch keine Garantie für ihre Genauigkeit oder für die Vergleichbarkeit zwischen verschiedenen JVM-Instanzen oder Systemen. Für `currentTimeMillis()` gilt diese Monotonie nicht zwingend, da Java die Zeit vom Betriebssystem bezieht und sich die Systemzeit ändern kann, etwa wenn die Uhr manuell angepasst wird. Differenzen von `currentTimeMillis()`-Zeitstempeln können dadurch komplett falsch und sogar negativ sein.¹



¹ Die Seite <http://stackoverflow.com/questions/351565/system-currenttimemillis-vs-system-nanotime> geht auf Details ein und verlinkt auf interne Implementierungen.

16.2.1 Profiler *

Wo die JVM im Programm überhaupt Taktzyklen verschwendet, zeigt ein *Profiler*. An diesen Stellen kann dann mit der Optimierung begonnen werden. *Java Mission Control* ist ein leistungsfähiges von Oracle bereitgestelltes Tool mit integriertem Profiler, das früher Teil des JDK war, inzwischen aber separat erhältlich ist. *Java VisualVM* ist ein weiteres freies Programm, das unter <https://visualvm.github.io/> bezogen werden kann. Auf der professionellen und kommerziellen Seite stehen sich *JProfiler* (<https://www.ej-technologies.com/products/jprofiler/overview.html>) und *YourKit* (<https://www.yourkit.com/java/profiler>) gegenüber. Die *Ultimate Version* von IntelliJ enthält ebenfalls einen Profiler.

16.3 Die Klasse Class

Angenommen, wir wollen einen Klassenbrowser schreiben. Dieser soll alle zum laufenden Programm gehörenden Klassen und darüber hinaus weitere Informationen anzeigen, wie Variablenbelegung, deklarierte Methoden, Konstruktoren und Informationen über die Vererbungshierarchie. Dafür benötigen wir die Bibliotheksklasse `Class`. Exemplare der Klasse `Class` sind Objekte, die etwa eine Java-Klasse, ein Record oder eine Java-Schnittstelle repräsentieren.

In diesem Punkt unterscheidet sich Java von vielen herkömmlichen Programmiersprachen, da sich Eigenschaften von Klassen vom gerade laufenden Programm mithilfe der `Class`-Objekte abfragen lassen. Bei den Exemplaren von `Class` handelt es sich um eine eingeschränkte Form von Meta-Objekten² – die Beschreibung eines Java-Typs, die aber nur ausgewählte Informationen preisgibt. Neben normalen Klassen werden auch Schnittstellen durch ein `Class`-Objekt repräsentiert und sogar Arrays und primitive Datentypen – statt `Class` wäre wohl der Klassenname `Type` passender gewesen.

16.3.1 An ein Class-Objekt kommen

Zunächst müssen wir für eine bestimmte Klasse das zugehörige `Class`-Objekt erfragen. `Class`-Objekte selbst kann nur die JVM erzeugen. Wir können das nicht (die Objekte sind immutabel, und der Konstruktor ist privat). Um einen Verweis auf ein `Class`-Objekt zu bekommen, bieten sich folgende Lösungen an:

- Ist ein Exemplar der Klasse verfügbar, rufen wir die `getClass()`-Methode des Objekts auf und erhalten das `Class`-Exemplar der zugehörigen Klasse.
- Jeder Typ enthält eine statische Variable mit dem Namen `.class` vom Typ `Class`, die auf das zugehörige `Class`-Exemplar verweist.

² Echte Metaklassen wären Klassen, deren jeweils einziges Exemplar die normale Java-Klasse ist. Dann wären etwa die normalen Klassenvariablen in Wahrheit Objektvariablen in der Metaklasse.

- ▶ Auch auf primitiven Datentypen ist das Ende `.class` erlaubt. Das gleiche Class-Objekt liefert die statische Variable `TYPE` der Wrapper-Klassen. Damit ist `int.class == Integer.TYPE`.
- ▶ Die statische Methode `Class.forName(String)` kann eine Klasse erfragen, und wir erhalten das zugehörige Class-Exemplar als Ergebnis. Ist der Typ noch nicht geladen, sucht und bindet `forName(String)` die Klasse ein. Weil das Suchen schiefgehen kann, ist eine `ClassNotFoundException` möglich.
- ▶ Haben wir bereits ein Class-Objekt, sind aber nicht an ihm, sondern an seinen Vorfahren interessiert, so können wir einfach mit `getSuperclass()` ein Class-Objekt für die Oberklasse erhalten.

Das folgende Beispiel zeigt drei Möglichkeiten auf, an ein Class-Objekt für `java.awt.Point` heranzukommen:

Listing 16.1 `src/main/java/com/tutego/insel/meta/GetClassObject.java`

```
Class<java.awt.Point> c1 = java.awt.Point.class;
IO.println( c1 );           // class java.awt.Point

Class<? extends java.awt.Point> c2 = new java.awt.Point().getClass();
IO.println( c2 );           // class java.awt.Point

try {
    Class<?> c3 = Class.forName( "java.awt.Point" );
    IO.println( c3 );       // class java.awt.Point
}
catch ( ClassNotFoundException e ) { e.printStackTrace(); }
```

Die Variante mit `forName(String)` ist sinnvoll, wenn der Name der gewünschten Klasse bei der Übersetzung des Programms noch nicht feststand. Sonst ist die vorhergehende Technik gängiger, und der Compiler kann prüfen, ob es den Typ gibt. Eine volle Qualifizierung ist nötig, `Class.forName("Point")` würde nur nach `Point` in dem Default-Paket suchen.

Beispiel

Klassenobjekte für primitive Elemente liefert `forName(String)` nicht! Die beiden Anweisungen `Class.forName("boolean")` und `Class.forName(boolean.class.getName())` führen zu einer `ClassNotFoundException`.

```
class java.lang.Object
```

- `final Class<? extends Object> getClass()`
Liefert zur Laufzeit das Class-Exemplar, das die Klasse des Objekts repräsentiert.



```
final class java.lang.Class<T>
implements Serializable, GenericDeclaration, Type, AnnotatedElement, Type-
Descriptor.OfField<Class<?>>, Constable
```

- `static Class<?> forName(String className)` throws `ClassNotFoundException`
Liefert das `Class`-Exemplar für die Klasse oder Schnittstelle mit dem angegebenen voll qualifizierten Namen. Falls der Typ bisher nicht vom Programm benötigt wurde, sucht und lädt der Klassenlader die Klasse. Die Methode liefert niemals null zurück. Falls die Klasse nicht geladen und eingebunden werden konnte, gibt es eine `ClassNotFoundException`. Eine alternative Methode `forName(String name, boolean initialize, ClassLoader loader)` ermöglicht auch das Laden mit einem gewünschten Klassenlader. Der Klassenname muss immer voll qualifiziert sein.

ClassNotFoundException und NoClassDefFoundError *

Eine `ClassNotFoundException` lösen die Methoden

- ▶ `forName(...)` aus `Class` und
- ▶ `loadClass(String name [, boolean resolve])` aus `ClassLoader` bzw.
- ▶ `findSystemClass(String name)` aus `ClassLoader`

immer dann aus, wenn der Klassenlader die Klasse nach ihrem Klassennamen nicht finden kann. Auslöser ist also die Anwendung, die dynamisch Typen laden will, die dann aber nicht vorhanden sind.

Neben der `ClassNotFoundException` gibt es einen `NoClassDefFoundError` – ein harter Linkage-Error, den die JVM immer dann auslöst, wenn sie eine im Bytecode referenzierte Klasse nicht laden kann. Nehmen wir zum Beispiel eine Anweisung wie `new MeineKlasse()`. Führt die JVM diese Anweisung aus, versucht sie, den Bytecode von `MeineKlasse` zu laden. Ist der Bytecode für `MeineKlasse` nach dem Compilieren entfernt worden, löst die JVM durch den nicht gelungenen Ladeversuch den `NoClassDefFoundError` aus. Auch tritt der Fehler auf, wenn beim Laden des Bytecodes die Klasse `MeineKlasse` zwar gefunden wurde, aber `MeineKlasse` einen statischen Initialisierungsblock besitzt, der wiederum eine Klasse referenziert, für die keine Klassendatei vorhanden ist.

Die Ausnahme `ClassNotFoundException` kommt häufiger vor als `NoClassDefFoundError` und ist im Allgemeinen ein Indiz dafür, dass ein Java-Archiv im Klassenpfad fehlt.

Probleme nach Anwendung eines Obfuscators *

Dass der Compiler automatisch Bytecode gemäß diesem veränderten Quellcode erzeugt, führt nur dann zu unerwarteten Problemen, wenn wir einen Obfuscator über den Programmtext laufen lassen, der nachträglich den Bytecode modifiziert, damit die Bedeutung des Programms bzw. des Bytecodes verschleiert und dabei Typen umbenennt. Offensichtlich darf

ein Obfuscator Typen, deren Class-Exemplare abgefragt werden, nicht umbenennen – oder der Obfuscator müsste die entsprechenden Zeichenketten ebenfalls korrekt ersetzen (aber natürlich nicht alle Zeichenketten, die zufällig mit Namen von Klassen übereinstimmen).

16.3.2 Eine Class ist ein Type

In Java gibt es unterschiedliche Typen, wobei Klassen, Records, Schnittstellen und Aufzählungstypen von der JVM als Class-Objekte repräsentiert werden. In der Reflection-API repräsentiert die Schnittstelle `java.lang.reflect.Type` alle Typen, und die einzige implementierende Klasse ist `Class`. Unter `Type` gibt es einige Unterschnittstellen:

- ▶ `ParameterizedType` (repräsentiert generische Typen wie `List<T>`)
- ▶ `TypeVariable<D>` (repräsentiert beispielsweise `T extends Comparable<? super T>`)
- ▶ `WildcardType` (repräsentiert etwa `? super T`)
- ▶ `GenericArrayType` (repräsentiert so etwas wie `T[]`)

Die einzige Methode von `Type` ist `getTypeName()`, und das ist sogar »nur« eine Default-Methode, die `toString()` aufruft.

`Type` ist die Rückgabe diverser Methoden in der Reflection-API, etwa von `getGenericSuperclass()` und `getGenericInterfaces()` der Klasse `Class` und von vielen weiteren Methoden, die die Javadoc unter »USE« aufzählt.

16.4 Die Utility-Klassen System und Properties

In der Klasse `java.lang.System` finden sich Methoden zum Erfragen und Ändern von Systemvariablen, zum Umlenken der Standarddatenströme, zum Ermitteln der aktuellen Zeit, zum Beenden der Applikation und noch für das eine oder andere. Alle Methoden sind ausschließlich statisch, und ein Exemplar von `System` lässt sich nicht anlegen. In der Klasse `java.lang.Runtime` finden sich zusätzlich Hilfsmethoden, wie für 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.

Bemerkung

Insgesamt machen die Klassen `System` und `Runtime` keinen besonders aufgeräumten Eindruck (siehe Abbildung 16.4); sie wirken so, als sei hier alles zu finden, was an anderer Stelle nicht mehr hineingepasst hat. Auch wären manche Methoden der einen Klasse genauso gut in der anderen Klasse aufgehoben.

Dass die statische Methode `System.arraycopy(...)` zum Kopieren von Arrays nicht in `java.util.Arrays` stationiert ist, lässt sich nur historisch erklären. Und `System.exit(int)` leitet an



`Runtime.getRuntime().exit(int)` weiter. Die Anzahl der Prozessoren liefert sowohl `Runtime.availableProcessors()` als auch ein MBean über `ManagementFactory.getOperatingSystemMXBean().getAvailableProcessors()`. Aber API-Design ist wie Sex: Eine unüberlegte Aktion, und die Brut lebt mit uns für immer.

java::lang::System	java::lang::Runtime
<pre> + err: PrintStream + in: InputStream + out: PrintStream + arraycopy(src: Object, srcPos: int, dest: Object, destPos: int, length: int) + clearProperty(key: String): String + console(): Console + currentTimeMillis(): long + exit(status: int) + gc() + getProperties(): Properties + getProperty(key: String, def: String): String + getProperty(key: String): String + getSecurityManager(): SecurityManager + getenv(name: String): String + getenv(): Map + identityHashCode(x: Object): int + inheritedChannel(): Channel + load(filename: String) + loadLibrary(libname: String) + mapLibraryName(libname: String): String + nanoTime(): long + runFinalization() + runFinalizersOnExit(Value: boolean) + setErr(err: PrintStream) + setIn(in: InputStream) + setOut(out: PrintStream) + setProperties(props: Properties) + setProperty(key: String, value: String): String + setSecurityManager(s: SecurityManager) </pre>	<pre> + addShutdownHook(hook: Thread) + availableProcessors(): int + exec(cmdarray: String[], envp: String[], dir: File): Process + exec(cmdarray: String[]): Process + exec(cmdarray: String[], envp: String[]): Process + exit(status: int) + freeMemory(): long + gc() + getLocalizedInputStream(in: InputStream): InputStream + getLocalizedOutputStream(out: OutputStream): OutputStream + getRuntime(): Runtime + halt(status: int) + load(filename: String) + loadLibrary(libname: String) + maxMemory(): long + removeShutdownHook(hook: Thread): boolean + runFinalization() + runFinalizersOnExit(value: boolean) + totalMemory(): long + traceInstructions(on: boolean) + traceMethodCalls(on: boolean) </pre>

Abbildung 16.4 Eigenschaften der Klassen »System« und »Runtime«

16.4.1 Speicher der JVM

Das `Runtime`-Objekt hat drei Methoden, die Auskunft über den Speicher der JVM geben:

- ▶ `maxMemory()` liefert die Anzahl der Bytes, die maximal für die JVM verfügbar sind. Der Wert kann beim Aufruf der JVM mit `-Xmx` in der Kommandozeile gesetzt werden.
- ▶ `totalMemory()` ist die aktuell vom Heap reservierte Größe und kann bis auf `maxMemory()` wachsen. Sie kann prinzipiell auch wieder schrumpfen. Es gilt: `maxMemory() > totalMemory()`.
- ▶ `freeMemory()` ist der Speicher, der innerhalb des aktuell zugewiesenen Heaps für neue Objekte verfügbar ist. Durch die automatische Speicherbereinigung kann dieser Wert wieder ansteigen. Es gilt: `totalMemory() > freeMemory()`. Allerdings ist `freeMemory()` nicht der gesamte freie verfügbare Speicherbereich, denn es fehlt noch der »Anteil« von `maxMemory()`.

Zwei nützliche Größen können berechnet werden.

Benutzter Speicher:

```
long usedMemory = Runtime.getRuntime().totalMemory() -
    Runtime.getRuntime(). freeMemory();
```

Gesamter noch verfügbarer Speicher (frei und noch reservierbar bis maxMemory()):

```
long totalFreeMemory = Runtime.getRuntime().maxMemory() - usedMemory;
```

Beispiel

Gib Informationen über den Speicher auf einem Rechner aus:

Listing 16.2 src/main/java/com/tutego/insel/locale/PrintMemory.java

```
long totalMemory    = Runtime.getRuntime().totalMemory();
long freeMemory     = Runtime.getRuntime().freeMemory();
long maxMemory      = Runtime.getRuntime().maxMemory();
long usedMemory     = totalMemory - freeMemory;
long totalFreeMemory = maxMemory - usedMemory;

System.out.printf(
    "total=%d MiB, free=%d MiB, max=%d MiB, used=%d MiB, total free=%d MiB%n",
    totalMemory >> 20, freeMemory >> 20, maxMemory >> 20,
    usedMemory >> 20, totalFreeMemory >> 20 );
```

Die Ausgabe kann sein:

```
total=256 MiB, free=246 MiB, max=4046 MiB, used=9 MiB, total free=4036 MiB
```

Für weitergehende Metriken lohnt sich zusätzlich ein Blick auf `MemoryMXBean` und `GarbageCollectorMXBean` oder ein Profiler.

[zB]

16.4.2 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:

```
IO.println( System.getProperty( "os.name" ) ); // z. B. Windows 11
```

Gib alle Systemeigenschaften auf dem Bildschirm aus:

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

[zB]

Die Ausgabe beginnt mit:

```
java.specification.version=25
sun.cpu.isalist=amd64
sun.jnu.encoding=Cp1252
java.class.path=C:\Users\Christian\AppData\Local\JetB...
java.vm.vendor=Eclipse Adoptium
```

Tabelle 16.2 zeigt eine Liste der wichtigsten Standardsystemeigenschaften:

Schlüssel	Bedeutung
java.version	Version der Java-Laufzeitumgebung
java.class.path	eigener 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 16.2 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` bzw. 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.

16.4.3 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 Weißraum) 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 16.3 `com/tutego/insel/lang/SetProperty.java`

```
Optional<String> logProperty = ofNullable( System.getProperty( "LOG" ) );
Optional<String> usernameProperty = ofNullable( System.getProperty( "USER" ) );
Optional<String> sizeProperty = ofNullable( System.getProperty( "SIZE" ) );

IO.println( logProperty.isPresent() ); // true
usernameProperty.ifPresent( IO::println ); // Chris
sizeProperty.map( Integer::parseInt ).ifPresent( IO::println ); // 100
IO.println( System.getProperty( "DEBUG", "false" ) ); // false
```

Wir bekommen über `getProperty(String)` 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 `null`-Test sagt also aus, ob `log-`

Property vorhanden ist oder nicht. Statt `-DLOG` führt auch `-DLOG=` zum gleichen Ergebnis, denn der assoziierte Wert ist der Leer-String. 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(String, String)`, der zwei Argumente übergeben werden, denn das zweite Argument steht für einen Default-Wert. So kann immer ein Standardwert angenommen werden.

Boolean.getBoolean(String)

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(String)` 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(String)` 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=false` ergeben immer `false`.³

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

■ `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. Der Vergleich mit `"true"` erfolgt ohne Beachtung der Groß-/Kleinschreibung. Gibt `false` zurück, wenn der Wert nicht `"true"` ist, nicht existiert oder `null` ist.

³ Das liegt an der Implementierung: `Boolean.valueOf("false")` liefert genauso `false` wie `Boolean.valueOf("")` oder `Boolean.valueOf(null)`.

16.4.4 Zeilenumbruchzeichen, `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 vieler ist die Darstellung des Zeilenumbruchs auf den gängigen Architekturen unterschiedlich:

- ▶ Unix und macOS: Line Feed (kurz LF), Zeilenvorschub, `\n`
- ▶ Windows: Carriage Return (kurz CR) und Line Feed

Der Steuercode für Carriage Return ist 13 (0x0D), der für Line Feed 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, also einen Seitenvorschub, spielt bei den Zeilenumbrüchen keine Rolle.)

In Java gibt es drei Möglichkeiten, an das Zeilenumbruchzeichen bzw. die Zeilenumbruchzeichenkette des Systems heranzukommen:

1. mit dem Aufruf von `System.getProperty("line.separator")`
2. mit dem Aufruf von `System.lineSeparator()`
3. Nicht immer ist es nötig, das Zeichen (genauer gesagt, eine mögliche Zeichenkette) einzeln zu erfragen. Ist das Zeichen Teil einer formatierten Ausgabe beim `Formatter`, `String.format(...)` bzw. `printf(...)`, so steht der Formatspezifizierer `%n` für genau die im System hinterlegte Zeilenumbruchzeichenkette.

16.4.5 Umgebungsvariablen des Betriebssystems

Die Systemeigenschaften der JVM werden von der JVM selbst bereitgestellt (z. B. `os.name`, `java.version`) und sind in der Regel plattformübergreifend standardisiert.

Daneben existieren die *Umgebungsvariablen* (engl. *environment variables*) des Betriebssystems. Sie stammen direkt vom Betriebssystem (z. B. `PATH`, `OS`) und können plattformspezifische, oft eher technische Werte enthalten. Fast jedes Betriebssystem nutzt dieses Konzept; bekannt ist etwa `PATH` für den Suchpfad von Programmen unter Windows und Unix.

Java ermöglicht den Zugriff auf diese Betriebssystem-Umgebungsvariablen über zwei statische Methoden:

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

- `static Map<String, String> getEnv()`
Liest eine Menge von `<String, String>`-Paaren mit allen Umgebungsvariablen des Betriebssystems, die dem aktuellen Java-Prozess zur Verfügung stehen.

■ `static String getenv(String name)`

Liest eine Umgebungsvariable mit dem Namen `name`. Gibt es sie nicht, ist die Rückgabe `null`.

Name der Variablen	Beschreibung	Beispiel
COMPUTERNAME	Name des Computers	<i>MOE</i>
HOMEDRIVE	Laufwerk des Benutzerverzeichnisses	<i>C:</i>
HOMEPATH	Pfad des Benutzerverzeichnisses	<i>\Users\Christian</i>
OS	Name des Betriebssystems*	<i>Windows_NT</i>
PATH	Suchpfad	<i>C:\windows\SYSTEM32; C:\windows ...</i>
PATHEXT	Dateiendungen, die für ausführbare Programme stehen	<i>.COM;.EXE;.BAT;.CMD;.VBS;.VBE; .JS;.JSE;.WSF;.WSH;.MSC</i>
SYSTEMDRIVE	Laufwerk des Betriebssystems	<i>C:</i>
TEMP und auch TMP	temporäres Verzeichnis	<i>C:\Users\CHRIST~1\AppData\Local\Temp</i>
USERDOMAIN	Domäne des Benutzers	<i>MOE</i>
USERNAME	Name des Nutzers	<i>Christian</i>
USERPROFILE	Profilverzeichnis	<i>C:\Users\Christian</i>
WINDIR	Verzeichnis des Betriebssystems	<i>C:\windows</i>

* Das Ergebnis weicht von `System.getProperty("os.name")` ab, das bei Windows 10 schon »Windows 10« liefert.

Tabelle 16.3 Auswahl einiger unter Windows verfügbarer Umgebungsvariablen

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



Beispiel

Lies die zugänglichen Umgebungsvariablen aus und gib sie aus:

```
Map<String, String> env= System.getenv();
env.forEach( (k, v) -> System.out.printf( "%s=%s\n", k, v ) );
```

16.5 Sprachen der Länder

Wenn Programme Ausgaben auf der Konsole oder auf einer grafischen Oberfläche erzeugen, sind sie oft fest mit einer bestimmten Landessprache verdrahtet. Ändert sich die Sprache, kann die Software nicht mit anderen landesüblichen Regeln etwa bei der Formatierung von Gleitkommazahlen umgehen. Dabei ist es gar nicht schwer, »mehrsprachige« Programme zu entwickeln, die unter verschiedenen Sprachen lokalisierte Ausgaben liefern. Im Grunde müssen wir alle sprachabhängigen Zeichenketten und Formatierungen von Daten durch Code ersetzen, der die landesüblichen Ausgaben und Regeln berücksichtigt. Java bietet hier eine Lösung an: zum einen durch die Definition einer Sprache, die dann Regeln vorgibt, nach denen die Java-API Daten automatisch formatieren kann, und zum anderen durch die Möglichkeit, sprachabhängige Teile in Ressourcendateien auszulagern.

16.5.1 Sprachen in Regionen über Locale-Objekte

In Java repräsentieren `Locale`-Objekte Sprachen in geografischen, politischen oder kulturellen 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. `Locale`-Objekte werden dann zum Beispiel einem `Formatter`, der hinter `String.format(...)` und `printf(...)` steht, oder einem `Scanner` übergeben. Diese Ausgaben nennen sich auf Englisch *locale-sensitive*.

Locale-Objekte aufbauen

`Locale`-Objekte werden immer mit dem Namen der Sprache und optional mit dem Namen des Landes bzw. einer Region und Variante erzeugt. Die `Locale`-Klasse bietet mehrere Möglichkeiten⁴ zum Aufbau der Objekte:

- ▶ drei `of(...)`-Methoden
- ▶ Die geschachtelte Klasse `Builder` von `Locale` nutzt das Builder-Pattern zum Aufbau neuer `Locale`-Objekte.
- ▶ über die `Locale`-Methode `forLanguageTag(...)` und eine String-Kennung

Beispiel

Bei den `of(...)`-Methoden der Klasse `Locale` werden Länderabkürzungen angegeben, etwa für ein Sprachobjekt für Großbritannien oder Frankreich:



⁴ `Locale`-Konstruktoren sind ab Java 19 deprecated.

```
Locale greatBritain = Locale.of( "en", "GB" );
Locale french       = Locale.of( "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 werden durch Zwei-Buchstaben-Kürzel aus dem ISO-639-Code⁵ (*ISO Language Code*) identifiziert, und die Ländernamen sind Zwei-Buchstaben-Kürzel, die in ISO 3166⁶ (*ISO Country Code*) beschrieben sind.



Beispiel

Drei Varianten zum Aufbau der `Locale.JAPANESE`:

```
Locale loc1 = Locale.of( "ja" );
Locale loc2 = new Locale.Builder().setLanguage( "ja" ).build();
Locale loc3 = Locale.forLanguageTag( "ja" );
```

Die `Locale`-Klasse hat weitere Methoden; für den Builder, für `forLanguageTag(...)` und die neuen Erweiterungen und Filtermethoden sollte die Javadoc studiert werden.⁷

Konstanten für einige Sprachen

Die `Locale`-Klasse besitzt Konstanten für häufig vorkommende Sprachen, optional mit Ländern. 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`. Manche Konstanten wirken dabei doppelt oder synonym. Die scheinbaren Dopplungen entstehen dadurch, dass `Locale` nicht nur verschiedene Sprachen unterscheidet, sondern diese zusätzlich mit Regionen oder Schriftsystemvarianten kombiniert.

Methoden, die `Locale`-Exemplare annehmen

`Locale`-Objekte sind als Objekte eigentlich uninteressant – sie haben Methoden, doch spannender ist der Typ als Identifikation für eine Sprache. In der Java-Bibliothek gibt es Dutzende Methoden, die `Locale`-Objekte annehmen und anhand deren ihr Verhalten anpassen. Beispiele sind `printf(Locale, ...)`, `format(Locale, ...)` und `toLowerCase(Locale)`.

⁵ https://de.wikipedia.org/wiki/Liste_der_ISO-639-1-Codes

⁶ <https://de.wikipedia.org/wiki/ISO-3166-1-Kodierliste>

⁷ Auf Englisch beschreibt das Java-Tutorial von Oracle die Erweiterungen unter <http://docs.oracle.com/javase/tutorial/i18n/locale/index.html>.



Tip

Gibt es keine Variante einer Formatierungs- bzw. Parse-Methode mit einem `Locale`-Objekt, unterstützt die Methode in der Regel kein sprachabhängiges Verhalten. Das Gleiche gilt für Objekte, die kein `Locale` über einen Konstruktor bzw. Setter annehmen. `Double.toString(...)` ist so ein Beispiel, auch `Double.parseDouble(...)`. In internationalisierten Anwendungen werden diese Methoden selten zu finden sein. Auch eine String-Konkatenation mit beispielsweise einer Gleitkommazahl ist nicht erlaubt (sie ruft intern eine `Double`-Methode auf), und ein `String.format(...)` ist allemal besser.

Methoden von `Locale` *

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



Beispiel

Gibt länderspezifische Sprachinformationen mithilfe statisch importierter `Locale`-Objekte aus:

Listing 16.4 `src/main/java/com/tutego/insel/locale/GermanyLocal.java`

```
IO.println(GERMANY.getCountry());           // DE
IO.println(GERMANY.getLanguage());          // de
IO.println(GERMANY.getVariant());           //
IO.println(GERMANY.getISO3Country());       // DEU
IO.println(GERMANY.getISO3Language());      // deu
IO.println(CANADA.getDisplayCountry());     // Kanada
IO.println(GERMANY.getDisplayLanguage());   // Deutsch
IO.println(GERMANY.getDisplayName());       // Deutsch (Deutschland)
IO.println(CANADA.getDisplayName());        // Englisch (Kanada)
IO.println(GERMANY.getDisplayName(FRENCH)); // allemand (Allemagne)
IO.println(CANADA.getDisplayName(FRENCH));  // anglais (Canada)
```

Es gibt auch statische Methoden zum Erfragen von `Locale`-Objekten:

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

- `static Locale getDefault()`
Liefert die von der JVM voreingestellte Sprache, die standardmäßig vom Betriebssystem stammt.

- `static Locale[] getAvailableLocales()`
Liefert eine Aufzählung aller installierten `Locale`-Objekte. Das Array enthält mindestens `Locale.US` und ca. 160 Einträge.
- `static String[] getISOCountries()`
Liefert ein Array mit allen aus zwei Buchstaben bestehenden ISO-3166-Country-Codes.
- `static Set<String> getISOCountries(Locale.IsoCountryCode type)`
Liefert eine Menge mit allen ISO-3166-Country-Codes, wobei die Aufzählung `IsoCountryCode` bestimmt: `PART1_ALPHA2` liefert den Code aus zwei Buchstaben, `PART1_ALPHA3` aus drei Buchstaben, `PART3` aus vier Buchstaben.

Zudem haben wir Methoden, die die Kürzel nach den ISO-Normen liefern:

```
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 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.
- `String getVariant()`
Liefert das Kürzel der Variante oder einen leeren String.

Diese Methoden geben zwar standardisierte Kürzel zurück, sind jedoch nicht für eine unmittelbar lesbare Darstellung vorgesehen. Für diverse `get*()`-Methoden gibt es entsprechende `getDisplay*()`-Methoden:

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

- `String getDisplayCountry(Locale inLocale)`
`final String getDisplayCountry()`
Liefert den Namen des Landes für Bildschirmausgaben für eine Sprache oder `Locale.getDefault()`.

- `String getDisplayLanguage(Locale inLocale)`
`final String getDisplayLanguage()`
Liefert den Namen der Sprache für Bildschirmausgaben für eine gegebene `Locale` oder `Locale.getDefault()`.
- `String getDisplayName(Locale inLocale)`
`final String getDisplayName()`
Liefert den Namen der Einstellungen für eine Sprache oder `Locale.getDefault()`.
- `String getDisplayVariant(Locale inLocale)`
`final String getDisplayVariant()`
Liefert den Namen der Variante für eine Sprache oder `Locale.getDefault()`.

16.6 Wichtige Datum-Klassen im Überblick

Da Datumsberechnungen schnell zu verschlungenen Konstrukten werden, ist es erfreulich, dass Java eine Vielzahl von Klassen zur Berechnung und Formatierung von Datums- und Zeitangaben bereitstellt. Diese Klassen sind bewusst abstrakt gehalten, sodass sie lokale Besonderheiten berücksichtigen können – etwa unterschiedliche Ausgabeformate, das Parsen von Datumsangaben, Zeitzonen oder die Umstellung zwischen Sommer- und Winterzeit in verschiedenen Kalendern.

Bis zur Java-Version 1.1 stand zur Darstellung und Manipulation von Datumswerten ausschließlich die Klasse `java.util.Date` zur Verfügung. Diese hatte mehrere Aufgaben:

- ▶ Erzeugung eines Datum/Zeit-Objekts aus Jahr, Monat, Tag, Minute und Sekunde
- ▶ Abfrage von Tag, Monat, Jahr ... mit der Genauigkeit von Millisekunden
- ▶ Ausgabe und Verarbeitung von Datum-Zeichenketten

Da die `Date`-Klasse nicht ganz fehlerfrei und internationalisiert war, wurden im JDK 1.1 neue Klassen eingeführt:

- ▶ `Calendar` nimmt sich der Aufgabe von `Date` an, zwischen verschiedenen Datumsrepräsentationen und Zeitskalen zu konvertieren. Die Unterklasse `GregorianCalendar` wird direkt erzeugt.
- ▶ `DateFormat` zerlegt Datum-Zeichenketten und formatiert die Ausgabe. Auch Datumsformate sind vom Land abhängig, das Java durch `Locale`-Objekte darstellt, und von einer Zeitzone, die durch die Exemplare der Klasse `TimeZone` repräsentiert ist.

In Java 8 zog eine weitere Datumsbibliothek mit ganz neuen Typen ein. Endlich können auch Datum und Zeit getrennt repräsentiert werden:

- ▶ `LocalDate`, `LocalTime`, `LocalDateTime` sind die temporalen Klassen für ein Datum, für eine Zeit und für eine Kombination aus Datum und Zeit.
- ▶ `Period` und `Duration` stehen für Abstände.

16.6.1 Der 1.1.1970

Der 1. Januar 1970 war ein Donnerstag mit wegweisenden Änderungen: In Großbritannien wurde die Volljährigkeit von 21 Jahren auf 18 Jahre herabgesetzt, und derselbe Zeitpunkt, 1. Januar 1970, 00:00:00 UTC, markiert in der Computergeschichte den Beginn der *Unix-Epoche*. In der *Unixzeit* wird die Zeit als Anzahl der vergangenen Sekunden relativ zu diesem Datum angegeben. Die Kennung *UTC* (*Coordinated Universal Time*) steht für die *koordinierte Weltzeit* und ist eine international standardisierte Zeit, die als Referenz für alle Zeitzonen dient. Sie basiert auf einer Kombination aus Atomzeit und astronomischer Zeitmessung. UTC ändert sich nie, da sie keine Sommer- oder Winterzeit kennt. Andere Zeitzonen werden als Abweichung von UTC angegeben, z. B.:

- ▶ Deutschland (MEZ – mitteleuropäische Zeit): UTC+1 (Winter)
- ▶ Deutschland (MESZ – mitteleuropäische Sommerzeit): UTC+2 (Sommer)

16.6.2 `System.currentTimeMillis()`

Auch für das Java-Entwicklungsteam ist die Unixzeit von Bedeutung, da viele Zeitstempel relativ zum 1. Januar 1970, 00:00:00 UTC sind. Die Methode `System.currentTimeMillis()` liefert die seit diesem Zeitpunkt vergangenen Millisekunden zurück. Dabei gilt:

- ▶ Die Rückgabe erfolgt als `long` (64 Bit), was für ca. 300 Millionen Jahre ausreicht.
- ▶ Die Messung hängt von der Systemuhr ab, die nicht immer millisekundengenau sein muss.
- ▶ Zeitzonen sind nicht berücksichtigt – das Ergebnis ist immer relativ zu UTC.

Soll die Zeit in der lokalen Zeitzone (z. B. deutsche Zeit) angezeigt werden, muss eine Umrechnung basierend auf UTC erfolgen.

16.6.3 Einfache Zeitumrechnungen durch `TimeUnit`

Eine Zeitdauer wird in Java oft durch Millisekunden ausgedrückt. 1.000 Millisekunden entsprechen einer Sekunde, 1.000×60 Millisekunden einer Minute usw. Diese ganzen großen Zahlen sind jedoch nicht besonders anschaulich, sodass zur Umrechnung `TimeUnit`-Objekte mit ihren `to*(...)`-Methoden genutzt werden. Java deklariert folgende Konstanten in `TimeUnit`: `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, `DAYS`, `HOURS`, `SECONDS`, `MINUTES`.

Jedes der Aufzählungskonstanten definiert die Umrechnungsmethoden `toDays(...)`, `toHours(...)`, `toMicros(...)`, `toMillis(...)`, `toMinutes(...)`, `toNanos(...)`, `toSeconds(...)`; sie bekommen ein `long` und liefern ein `long` in der entsprechenden Einheit. Zudem gibt es zwei `convert(...)`-Methoden, die von einer Einheit in eine andere umrechnen.



Beispiel

Konvertiere 23.746.387 Millisekunden in Stunden:

```
int v = 23_746_387;
IO.println( TimeUnit.MILLISECONDS.toHours( v ) ); // 6
IO.println( TimeUnit.HOURS.convert( v, TimeUnit.MILLISECONDS ) ); // 6
```

Sowohl `toHours(...)` als auch `convert(...)` geben nur die vollen Stunden zurück, und Bruchteile einer Stunde werden abgeschnitten, nicht gerundet.

```
enum java.util.concurrent.TimeUnit
extends Enum<TimeUnit>
implements Serializable, Comparable<TimeUnit>
```

- NANoseconds, MICROseconds, MILLIseconds, SECONDS, MINUTES, HOURS, DAYS

Aufzählungskonstanten von `TimeUnit`

- `long toDays(long duration)`
- `long toHours(long duration)`
- `long toMicros(long duration)`
- `long toMillis(long duration)`
- `long toMinutes(long duration)`
- `long toNanos(long duration)`
- `long toSeconds(long duration)`

- `long convert(long sourceDuration, TimeUnit sourceUnit)`

Liefert `sourceUnit.to*(sourceDuration)`, wobei * für die jeweilige Einheit steht. Beispielsweise liefert es `HOURS.convert(sourceDuration, sourceUnit)`, dann `sourceUnit.toHours(1)`. Die Lesbarkeit der Methode ist nicht optimal, daher sollten die anderen Methoden bevorzugt werden. Ergebnisse werden unter Umständen abgeschnitten, nicht gerundet. Gibt es einen Überlauf, folgt keine `ArithmeticException`.

- `long convert(Duration duration)`

Konvertiert die übergebene `duration` in die Zeiteinheit, die die aktuelle `TimeUnit` repräsentiert. So liefert `TimeUnit.MINUTES.convert( Duration.ofHours(12) )` zum Beispiel 720. Damit sind etwa `aunit.convert(Duration.ofNanos(n))` und `aunit.convert(n, NANoseconds)` gleich.

16.7 Date-Time-API

Das Paket `java.time` basiert auf dem standardisierten Kalendersystem von ISO-8601, und das deckt ab, wie ein Datum, wie Zeit, Datum und Zeit, UTC, Zeitintervalle (Dauer/Zeitspanne)

und Zeitzonen repräsentiert werden. Die Implementierung basiert auf dem gregorianischen Kalender, wobei auch andere Kalendertypen denkbar sind. Javas Kalendersystem greift auf andere Standards bzw. Implementierungen zurück, unter anderem auf das *Unicode Common Locale Data Repository* (CLDR) zur Lokalisierung von Wochentagen oder auf die *Time-Zone Database* (TZDB), die alle Zeitzonewechsel seit 1970 dokumentiert.

16.7.1 Erster Überblick

Die zentralen temporalen Typen aus der Date-Time-API sind schnell dokumentiert:

Typ	Beschreibung	Feld(er)
LocalDate	Repräsentiert ein übliches Datum.	Jahr, Monat, Tag
LocalTime	Repräsentiert eine übliche Zeit.	Stunden, Minuten, Sekunden, Nanosekunden
LocalDateTime	Kombination aus Datum und Zeit	Jahr, Monat, Tag, Stunden, Minuten, Sekunden, Nanosekunden
Period	Dauer zwischen zwei Local-Dates	Jahr, Monat, Tag
Year	nur Jahr	Jahr
Month	nur Monat	Monat
MonthDay	nur Monat und Tag	Monat, Tag
OffsetTime	Zeit mit Zeitzone	Stunden, Minuten, Sekunden, Nanosekunden, Zonen-Offset
OffsetDateTime	Datum und Zeit mit Zeitzone als UTC-Offset	Jahr, Monat, Tag, Stunden, Minuten, Sekunden, Nanosekunden, Zonen-Offset
ZonedDateTime	Datum und Zeit mit Zeitzone als ID und Offset	Jahr, Monat, Tag, Stunden, Minuten, Sekunden, Nanosekunden, Zonen-Info
Instant	Zeitpunkt (fortlaufende Maschinenzeit)	Nanosekunden
Duration	Zeitintervall zwischen zwei Instants	Sekunden/Nanosekunden

Tabelle 16.4 Alle temporalen Klassen aus »java.time«

16.7.2 Menschenzeit und Maschinenzeit

Datum und Zeit, die wir als Menschen in Einheiten wie Tagen und Minuten verstehen, nennen wir *Menschenzeit* (engl. *human time*). Die fortlaufende Zeit des Computers, die eine Auflösung bis in den Nanosekundenbereich haben kann, nennen wir *Maschinenzeit*. In der Date-Time-API startet die Maschinenzeit immer von der Unix-Epoche, also dem 1. Januar 1970, 00:00:00 UTC.

Die meisten Klassen sind für die Darstellung von Menschenzeit konzipiert; nur `Instant` und `Duration` beziehen sich auf die Maschinenzeit. `LocalDate`, `LocalTime` und `LocalDateTime` repräsentieren Menschenzeit ohne Bezug zu einer Zeitzone, `ZonedDateTime` dagegen mit Zeitonenbezug.

Bei der Auswahl der richtigen Zeitklassen ist natürlich die erste Überlegung, ob die Menschenzeit oder die Maschinenzeit repräsentiert werden soll. Dann folgen die Fragen, was genau für Felder nötig sind und ob eine Zeitzone relevant ist oder nicht. Soll zum Beispiel die Ausführungszeit gemessen werden, ist es unnötig, zu wissen, an welchem Datum die Messung beginnt und endet; hier ist `Duration` korrekt, nicht `Period`.

Beispiel

```

    LocalDate now = LocalDate.now();
    IO.println( now ); // z. B. 2026-01-31
    System.out.printf( "%d. %s %d%n",
                       now.getDayOfMonth(), now.getMonth(), now.getYear() );
    // z. B. 31. JANUARY 2026
    LocalDate bdayMLKing = LocalDate.of( 1929, Month.JANUARY, 15 );
    DateTimeFormatter formatter =
        DateTimeFormatter.ofLocalizedDate( FormatStyle.MEDIUM );
    IO.println( bdayMLKing.format( formatter ) ); // 15. Januar 1929
    Die Methode getMonth() auf einem LocalDate liefert als Ergebnis ein java.time.Month-
    Objekt, und das sind Aufzählungen. Die toString()-Repräsentation liefert die Konstante in
    Großbuchstaben.

```



Alle Klassen basieren standardmäßig auf dem ISO-System. Andere Kalendersysteme, wie der japanische Kalender, werden über Typen aus `java.time.chrono` erzeugt, und natürlich sind auch ganz neue Systeme möglich.

Beispiel

Ausgabe beim japanischen Kalender:

```

    ChronoLocalDate now = JapaneseChronology.INSTANCE.dateNow();
    IO.println( now ); // Japanese Reiwa 4-01-31

```



Paketübersicht

Die Typen der Date-Time-API verteilen sich auf verschiedene Pakete:

- ▶ `java.time`: Enthält die Standardklassen wie `LocalTime` und `Instant`. Alle Typen basieren auf dem Kalendersystem ISO-8601, das landläufig unter »gregorianischer Kalender« bekannt ist. Dieser wird zum sogenannten *Proleptic Gregorian Calendar* erweitert. Das ist ein gregorianischer Kalender, der auch für die Zeit vor 1582 (der Einführung dieses Kalenders) gültig ist, damit eine konsistente Zeitlinie entsteht.
- ▶ `java.time.chrono`: Hier befinden sich vorgefertigte alternative (also Nicht-ISO-)Kalendersysteme, wie der japanische Kalender, der Thai-Buddhist-Kalender, der islamische Kalender und ein paar weitere.
- ▶ `java.time.format`: Klassen zum Formatieren und Parsen von Datums- und Zeitwerten, wie der genannte `DateTimeFormatter`
- ▶ `java.time.zone`: unterstützende Klassen für Zeitzonen, etwa `ZonedDateTime`
- ▶ `java.time.temporal`: tiefer liegende API, die Zugriff und Modifikation einzelner Felder eines Datums-/Zeitwerts erlaubt

Designprinzipien

Bevor wir uns mit den einzelnen Klassen auseinandersetzen, wollen wir uns mit den Designprinzipien beschäftigen, denn alle Typen der Date-Time-API folgen wiederkehrenden Mustern. Die erste und wichtigste Eigenschaft ist, dass alle Objekte *immutable* sind, also nicht veränderbar. Das ist bei der »alten« API anders: `Date` und die `Calendar`-Klassen sind veränderbar, mit teils verheerenden Folgen. Denn werden diese Objekte herumgereicht und verändert, kann es zu unkalkulierbaren Seiteneffekten kommen. Die Klassen der neuen Date-Time-API sind *immutable*, und so stehen die Datum/Zeit-Klassen wie `LocalTime` oder `Instant` den veränderbaren Typen wie `Date` oder `Calendar` gegenüber. Alle Methoden, die nach Änderung aussehen, erzeugen neue Objekte mit den gewünschten Änderungen. Seiteneffekte bleiben also aus, und alle Typen sind threadsicher.

Unveränderbarkeit ist eine Designeigenschaft wie auch die Tatsache, dass `null` nicht als Argument erlaubt wird. In der Java-API wird oftmals `null` akzeptiert, weil es etwas Optionales ausdrückt, doch die Date-Time-API straft dies in der Regel mit einer `NullPointerException`. Dass `null` nicht als Argument und nicht als Rückgabe im Einsatz ist, kommt einer weiteren Eigenschaft zugute: Der Code wird über eine Fluent-API, also kaskadierte Ausdrücke, geschrieben, da viele Methoden die `this`-Referenz zurückgeben, so wie das auch von `StringBuilder` bekannt ist.

Zu diesen eher technischen Eigenschaften kommt die konsistente Namensgebung hinzu, die sich von der Namensgebung der bekannten JavaBeans absetzt. So gibt es keine Konstruktoren und keine Setter (das brauchen die *immutable* Klassen nicht), sondern Muster, die viele der Typen aus der Date-Time-API einhalten:

Methode	Klassen-/Exemplarmethode	grundsätzliche Bedeutung
<code>now()</code>	statisch	Liefert ein Objekt mit aktueller Zeit/aktuellem Datum.
<code>of*()</code>	statisch	Erzeugt neue Objekte.
<code>from*()</code>	statisch	Erzeugt neue Objekte aus anderen Repräsentationen.
<code>parse*()</code>	statisch	Erzeugt ein neues Objekt aus einer String-Repräsentation.
<code>format()</code>	Exemplar	Formatiert und liefert einen String.
<code>get*()</code>	Exemplar	Liefert Felder eines Objekts.
<code>is*()</code>	Exemplar	Fragt den Status eines Objekts ab.
<code>with*()</code>	Exemplar	Liefert eine Kopie des Objekts mit einer geänderten Eigenschaft.
<code>plus*()</code>	Exemplar	Liefert eine Kopie des Objekts mit einer aufsummierten Eigenschaft.
<code>minus*()</code>	Exemplar	Liefert eine Kopie des Objekts mit einer reduzierten Eigenschaft.
<code>to*()</code>	Exemplar	Konvertiert ein Objekt in einen neuen Typ.
<code>at*()</code>	Exemplar	Kombiniert dieses Objekt mit einem anderen Objekt.
<code>*Into()</code>	Exemplar	Kombiniert ein eigenes Objekt mit einem anderen Zielobjekt.

Tabelle 16.5 Namensmuster in der Date-Time-API

Die Methode `now()` haben wir schon in den ersten Beispielen verwendet, sie liefert zum Beispiel das aktuelle Datum. Weitere Erzeugermethoden sind die mit dem Präfix `of`, `from` oder `with`; Konstruktoren gibt es nicht. Die Methoden nach der Bauart `with*()` nehmen die Rolle der Setter ein.

16.7.3 Die Datumsklasse `LocalDate`

Ein Datum (ohne Zeitzone) repräsentiert die Klasse `LocalDate`. Damit lässt sich zum Beispiel ein Geburtsdatum repräsentieren.

Ein temporales Objekt kann über die statischen `of(...)`-Fabrikmethode aufgebaut und über `ofInstant(Instant instant, ZoneId zone)` oder von einem anderen temporalen Objekt abgeleitet werden. Interessant sind die Methoden, die mit einem `TemporalAdjuster` arbeiten.

Mit den Objekten in der Hand können wir diverse Getter nutzen und einzelne Felder erfragen, etwa `getDayOfMonth()`, `getDayOfYear()` (liefern `int`) oder `getDayOfWeek()`, das eine Aufzählung vom Typ `DayOfWeek` liefert, und `getMonth()`, das eine Aufzählung vom Typ `Month` liefert. Des Weiteren gibt es `long toEpochDay()` und `long toEpochSecond(LocalTime time, ZoneOffset offset)`.



Beispiel

Finde von heute aus gesehen den nächsten Samstag:

```
LocalDate today = LocalDate.now();
LocalDate nextSaturday = today.with( TemporalAdjusters.next(DayOfWeek.SATURDAY) );
System.out.printf( "Today is %s, and next Saturday is %s",
                    today, nextSaturday );
```

Dazu kommen Methoden, die mit `minus*()` oder `plus*()` neue `LocalDate`-Objekte liefern, wenn zum Beispiel mit `minusYears(long yearsToSubtract)` eine Anzahl Jahre zurückgelaufen werden soll. Durch die Negation des Vorzeichens kann auch die jeweils entgegengesetzte Methode genutzt werden, sprich, `LocalDate.now().minusMonths(1)` kommt zum gleichen Ergebnis wie `LocalDate.now().plusMonths(-1)`. Die `with*()`-Methoden belegen ein Feld neu und liefern ein modifiziertes neues `LocalDate`-Objekt.

Von einem `LocalDate` lassen sich andere temporale Objekte bilden; `atTime(...)` etwa liefert `LocalDateTime`-Objekte, bei denen gewisse Zeitfelder belegt sind. `atTime(int hour, int minute)` ist so ein Beispiel. Mit `until(...)` lässt sich eine Zeitdauer vom Typ `Period` liefern. Interessant sind zwei Methoden, die einen Strom von `LocalDate`-Objekten bis zu einem Endpunkt liefern:

- `Stream<LocalDate> datesUntil( LocalDate endExclusive )`
- `Stream<LocalDate> datesUntil( LocalDate endExclusive, Period step )`

16.8 Logging mit Java

Das Loggen (Protokollieren) von Informationen über Programmzustände ist ein wichtiger Teil, um später den Ablauf und die Zustände von Programmen rekonstruieren und verstehen zu können. Mit einer Logging-API lassen sich Meldungen auf die Konsole oder in externe Speicher wie Text- bzw. XML-Dateien und Datenbanken schreiben oder über einen Chat verbreiten.

16.8.1 Logging-APIs

In der Java-Welt existieren bis heute mehrere konkurrierende Logging-APIs und -Implementierungen. Da die Java-Standardbibliothek in den ersten Versionen keine Logging-API enthielt, füllte die Open-Source-Bibliothek *Log4j* (später Log4j 2) früh diese Lücke und wurde in vielen Projekten zum Standard. Mit Java 1.4 hielt dann die Logging-API nach JSR 47 Einzug (`java.util.logging`, kurz *JUL*). Sie war jedoch weder API-kompatibel mit Log4j noch in allen Aspekten so leistungsfähig, was zu einer fragmentierten Landschaft führte.

Heute hat sich das Bild weiterentwickelt: In vielen Projekten wird JUL weiterhin genutzt, insbesondere wenn externe Abhängigkeiten vermieden werden sollen oder der Funktionsumfang ausreicht. In größeren Projekten kommt jedoch häufig ein einheitlicher Logging-API-Ansatz zum Einsatz, allen voran *SLF4J* als Abstraktionsschicht. SLF4J ermöglicht es, zur Laufzeit zwischen Implementierungen wie Log4j 2, JUL oder Logback zu wählen und so Mehrfachkonfigurationen zu vermeiden.

In diesem Text werden wir uns auf JUL konzentrieren und keine externe Bibliothek verwenden.

16.8.2 Logging mit `java.util.logging`

Mit der Java-Logging-API lässt sich eine Meldung schreiben, die sich dann zur Wartung oder zur Sicherheitskontrolle einsetzen lässt. Die API ist einfach:

Listing 16.5 `src/main/java/com/tutego/insel/logging/LoggerDemo.java`

```
package com.tutego.isel.logging;

import static java.time.temporal.ChronoUnit.MILLIS;
import static java.time.Instant.now;
import java.time.Instant;
import java.util.logging.Level;
import java.util.logging.Logger;

public class LoggerDemo {

    private static final Logger log = Logger.getLogger( LoggerDemo.class.getName() );

    static void main() {
        Instant start = now();
        log.info( "About to start" );

        try {
            log.log( Level.INFO, "Let's try to throw {0}", "null" );
            throw null;
        }
    }
}
```

```
    }  
    catch ( Exception e ) {  
        log.log( Level.SEVERE, "Oh Oh", e );  
    }  
    log.info( () -> String.format( "Runtime: %s ms",  
                                   start.until(now(), MILLIS )) );  
}  
}
```

Lassen wir das Beispiel laufen, folgt auf der Konsole die Warnung:

```
Juli 13, 2025 1:40:15 PM com.tutego.insel.logging.LoggerDemo main  
INFORMATION: About to start  
Juli 13, 2025 1:40:15 PM com.tutego.insel.logging.LoggerDemo main  
INFORMATION: Let's try to throw null  
Juli 13, 2025 1:40:15 PM com.tutego.insel.logging.LoggerDemo main  
SCHWERWIEGEND: Oh Oh  
java.lang.NullPointerException: Cannot throw exception because "null" is null  
    at com.tutego.insel.logging.LoggerDemo.main(LoggerDemo.java:20)  
  
Juli 13, 2025 1:40:15 PM com.tutego.insel.logging.LoggerDemo main  
INFORMATION: Runtime: 229 ms
```

Das Logger-Objekt

Zentrales Element ist ein Logger-Objekt, das in der Regel mit `Logger.getLogger(String name)` erstellt wird, wobei `name` üblicherweise dem voll qualifizierten Klassennamen entspricht. Diese Variante wird bevorzugt, da der Logger so eindeutig einer Klasse oder einem Namen zugeordnet werden kann und sich leichter konfigurieren lässt. Seltener wird `Logger.getAnonymousLogger()` verwendet, das einen namenlosen Logger ohne Klassenbezug erzeugt, was die Konfiguration erschwert.

Oft ist der Logger als `private` (statische) finale Variable in der Klasse deklariert.

Loggen mit Log-Level

Nicht jede Meldung ist gleich wichtig. Einige sind für das Debuggen oder wegen der Zeitmessungen hilfreich, doch Ausnahmen in den `catch`-Zweigen sind enorm wichtig. Damit verschiedene Detailgrade unterstützt werden, lässt sich ein *Log-Level* festlegen. Er bestimmt, wie »ernst« der Fehler bzw. eine Meldung ist. Das ist später wichtig, wenn die Fehler nach ihrer Dringlichkeit aussortiert werden. Die Log-Level sind in der Klasse `Level` als Konstanten deklariert:⁸

⁸ Da das Logging-Framework in Version 1.4 zu Java stieß, nutzt es noch keine typisierten Aufzählungen, denn die gibt es erst seit Java 5.

- ▶ **FINEST (kleinste Stufe)**
- ▶ FINER
- ▶ FINE
- ▶ CONFIG
- ▶ INFO
- ▶ WARNING
- ▶ **SEVERE (höchste Stufe)**

Zum Loggen selbst bietet die `Logger`-Klasse die allgemeine Methode `log(Level level, String msg)` bzw. für jeden Level eine eigene Methode:

Level	Aufruf über <code>log(...)</code>	Spezielle Log-Methode
SEVERE	<code>log(Level.SEVERE, msg)</code>	<code>severe(String msg)</code>
WARNING	<code>log(Level.WARNING, msg)</code>	<code>warning(String msg)</code>
INFO	<code>log(Level.INFO, msg)</code>	<code>info(String msg)</code>
CONFIG	<code>log(Level.CONFIG, msg)</code>	<code>config(String msg)</code>
FINE	<code>log(Level.FINE, msg)</code>	<code>fine(String msg)</code>
FINER	<code>log(Level.FINER, msg)</code>	<code>finer(String msg)</code>
FINEST	<code>log(Level.FINEST, msg)</code>	<code>finest(String msg)</code>

Tabelle 16.6 Log-Level und Methoden

Alle diese Methoden setzen eine Mitteilung vom Typ `String` ab. Sollen eine Ausnahme und der dazugehörige Stack-Trace geloggt werden, muss zu folgender Logger-Methode gegriffen werden, die auch schon das Beispiel nutzt:

■ `void log(Level level, String msg, Throwable thrown)`

Die Varianten von `severe(...)`, `warning(...)` usw. sind nicht überladen mit einem Parametertyp `Throwable`.

16.9 Maven: Build-Management und Abhängigkeiten auflösen

Im ersten Kapitel haben wir schon ein Maven-Projekt angelegt, allerdings noch nie wirklich von Maven profitiert. Zwei Dinge stechen hervor:

1. Abhängigkeiten können einfach deklariert werden, und sie werden von Maven automatisch heruntergeladen, auch inklusive aller Unterabhängigkeiten. Die besondere Stärke von Maven liegt im Auflösen transitiver Abhängigkeiten.
2. Der Build: Java-Quellcode alleine macht noch kein Projekt aus; die Quellen müssen übersetzt werden, Testfälle müssen laufen, die Javadoc sollte generiert werden, am Ende steht in aller Regel eine komprimierte JAR-Datei.

16.9.1 Dependency hinzunehmen

Wir wollen als Beispiel eine Abhängigkeit zu dem kleinen Web-Framework *Spark* (<https://sparkjava.com/>) herstellen. Öffnen wir *pom.xml* und ergänzen wir das Fettgedruckte für die Abhängigkeit:

Listing 16.6 pom.xml

```
<project ...>
...
<properties>
  <maven.compiler.release>25</maven.compiler.release>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencies>
  <dependency>
    <groupId>com.sparkjava</groupId>
    <artifactId>spark-core</artifactId>
    <version>2.9.4</version>
  </dependency>
</dependencies>
</project>
```

Alle Abhängigkeiten befinden sich in einem speziellen XML-Element `<dependencies>`. Darunter finden sich dann beliebig viele `<dependency>`-Blöcke.

Alles ist vorbereitet, Zeit für das Hauptprogramm:

Listing 16.7 src/main/java/SparkServer.java

```
static void main() {
  spark.Spark.get( "/hello", ( req, _ ) -> "Hello Browser " + req.userAgent() );
}
```

Starten wir das Programm wie üblich, startet ein Webserver, und unter der URL *http://localhost:4567/hello* können wir die Ausgabe ablesen. (Die Logger-Ausgaben können wir ignorieren.)

16.9.2 Lokales und das Remote Repository

Das Auflösen der abhängigen Java-Archive dauert beim ersten Mal länger, da Maven ein Remote Repository kontaktiert und von dort immer die neuesten JAR-Dateien bezieht und lokal ablegt. Das umfangreiche Remote Repository speichert zu vielen bekannten quelloffenen Projekten fast alle Versionen von JAR-Dateien. Das *Central Repository* hat die URL <https://repo.maven.apache.org/maven2/>. Dieses Central Repository ist die Standardquelle, aber Projekte können auch andere Repositories konfigurieren.

IntelliJ bezieht standardmäßig nicht die Ressourcen selbstständig, sondern das muss angestoßen werden. Dazu erscheint im Editor oben rechts ein kleines Rädchen mit einem M, auf das gedrückt werden muss.



Gespeichert werden die heruntergeladenen Ressourcen selbst nicht im Projekt, sondern in einem lokalen Repository, das im Heimatverzeichnis des Anwenders liegt und *.m2* heißt. Auf diese Weise teilen sich alle Maven-Projekte die gleichen JAR-Dateien, und diese müssen nicht projektweise immer neu bezogen und aktualisiert werden.

16.9.3 Lebenszyklus, Phasen und Maven-Plugins

Maven definiert drei Hauptlebenszyklen: *clean*, *default* und *site*. Innerhalb dieser Zyklen gibt es *Phasen*, zum Beispiel in *default* die Phase *compile* zum Übersetzen der Quellen. Alles, was Maven ausführt, sind *Plugins*, etwa *compiler* und viele andere, die <https://maven.apache.org/plugins/> auflistet. Ein Plugin kann unterschiedliche *Goals* ausführen. So kennt zum Beispiel das Javadoc-Plugin (beschrieben unter <https://maven.apache.org/components/plugins/maven-javadoc-plugin/>) aktuell 16 Goals. Ein Goal wird später über die Kommandozeile angesprochen oder über die IDE.

Ein Java-Archiv wird zum Beispiel über die *package*-Phase erzeugt:

```
$ mvn package
```

Das Kommandozeilenwerkzeug muss im gleichen Verzeichnis aufgerufen werden, in dem auch die POM-Datei steht.

16.10 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 dem Aufbau und der Konfiguration von Java-Projekten, objektrelationalen Mappern (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 (<https://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 (<https://gpl-violations.org/>) werden öffentlich angeprangert und sind unange-nehm. Wer in Java entwickelt, sollte für den kommerziellen Vertrieb sein Augenmerk ver-stärkt 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 *Apache Commons* (<http://commons.apache.org/>) eine hübsche Sammlung an Klassen und Methoden zusammengetra-gen, und das Studium der Quellen sollte für Softwareentwickler mehr zum Alltag gehören. Die Website <https://www.openhub.net/> eignet sich dafür außerordentlich gut, da sie eine Su-che über bestimmte Stichwörter durch mehr als eine Milliarde Quellcodezeilen verschiede-ner Programmiersprachen ermöglicht; erstaunlich, wie viele »F*ck« schreiben. Und »Porn Groove« kannte ich vor dieser Suche auch noch nicht.

Auf einen Blick

1	Java ist auch eine Sprache	49
2	Imperative Sprachkonzepte	91
3	Klassen und Objekte	221
4	Arrays und ihre Anwendungen	265
5	Der Umgang mit Zeichen und Zeichenketten	311
6	Eigene Klassen schreiben	395
7	Objektorientierte Beziehungsfragen	459
8	Records, Schnittstellen, Aufzählungen, versiegelte Klassen	523
9	Ausnahmen müssen sein	597
10	Geschachtelte Typen	661
11	Besondere Typen der Java SE	681
12	Generics<T>	751
13	Lambda-Ausdrücke und funktionale Programmierung	807
14	Architektur, Design und angewandte Objektorientierung	877
15	Java Platform Module System	891
16	Die Klassenbibliothek	913
17	Einführung in die nebenläufige Programmierung	947
18	Einführung in Datenstrukturen und Algorithmen	983
19	Einführung in grafische Oberflächen	1039
20	Einführung in Dateien und Datenströme	1049
21	Einführung ins Datenbankmanagement mit JDBC	1087
22	Bits und Bytes, Mathematisches und Geld	1093
23	Testen mit JUnit	1153
24	Die Werkzeuge des JDK	1179

Inhalt

Materialien zum Buch	32
Vorwort	33

1 Java ist auch eine Sprache 49

1.1 Historischer Hintergrund	50
1.1.1 Wo ist die Sonne? Oracle übernimmt Sun Microsystems 2010	51
1.2 Warum Java populär ist – die zentralen Eigenschaften	52
1.2.1 Bytecode	52
1.2.2 Ausführung des Bytecodes durch eine virtuelle Maschine	53
1.2.3 Plattformunabhängigkeit	53
1.2.4 Java als Sprache, Laufzeitumgebung und Standardbibliothek	53
1.2.5 Objektorientierung in Java	54
1.2.6 Java ist verbreitet und bekannt	54
1.2.7 Java ist schnell – Optimierung und Just-in-time-Compilation	55
1.2.8 Zeiger und Referenzen	57
1.2.9 Bring den Müll raus, Garbage-Collector!	58
1.2.10 Ausnahmebehandlung	59
1.2.11 Das Angebot an Bibliotheken und Werkzeugen	60
1.2.12 Vergleichbar einfache Syntax	60
1.2.13 Verzicht auf umstrittene Konzepte	61
1.2.14 Java ist Open Source	62
1.3 Java im Vergleich zu anderen Sprachen *	63
1.3.1 Java und C(++)	63
1.3.2 Java und JavaScript	64
1.3.3 Java und C#/.NET	64
1.4 Einschränkungen von Java	65
1.4.1 Wofür sich Java weniger eignet	65
1.4.2 Alternative Sprachen und Technologien	66
1.4.3 Brücken zur Systemnähe in Java	66
1.5 Weiterentwicklung und Umbrüche	67
1.5.1 Die Entwicklung von Java und seine Zukunftsaussichten	67
1.5.2 Features, Enhancements (Erweiterungen) und ein JSR	68
1.5.3 Applets	69
1.5.4 JavaFX	69

1.6	Java-Plattformvarianten	70
1.6.1	Die Java SE-Plattform	70
1.6.2	Java ME: Java für die Kleinen	73
1.6.3	Java für die ganz, ganz Kleinen	74
1.6.4	Jakarta EE: Java für die Großen	74
1.7	Java SE-Implementierungen	75
1.7.1	OpenJDK	75
1.7.2	Oracle JDK	76
1.8	Installation des JDK und erster Start	78
1.8.1	Oracle JDK unter Windows installieren	78
1.8.2	Das erste Programm compilieren und testen	80
1.8.3	Der Compilerlauf	81
1.8.4	Die Laufzeitumgebung	82
1.8.5	Häufige Compiler- und Interpreter-Probleme	82
1.9	Entwicklungsumgebungen	84
1.9.1	IntelliJ IDEA	84
1.9.2	Eclipse IDE	88
1.9.3	NetBeans	90
1.10	Zum Weiterlesen	90

2 Imperative Sprachkonzepte 91

2.1	Von der Klasse zur Anweisung	91
2.1.1	Was sind Anweisungen?	92
2.1.2	Die Reise beginnt am main	92
2.1.3	Der erste Methodenaufruf: println(...)	94
2.1.4	(Reservierte) Schlüsselwörter	95
2.1.5	Bezeichner	97
2.1.6	Literale	99
2.1.7	Atomare Anweisungen und Anweisungssequenzen	100
2.1.8	Mehr zu print(...), println(...) und printf(...) für Bildschirmausgaben	100
2.1.9	Die API-Dokumentation	103
2.1.10	Ausdrücke	105
2.1.11	Ausdrucksanweisung	105
2.1.12	Erster Einblick in die Objektorientierung	106
2.1.13	Modifizierer	107
2.1.14	Gruppieren von Anweisungen mit Blöcken	107

2.2	Lexikalische Grundlagen	109
2.2.1	Tokens	109
2.2.2	Textkodierung durch Unicode-Zeichen	110
2.2.3	Zusammenfassung der lexikalischen Analyse	111
2.2.4	Kommentare	111
2.3	Datentypen, Typisierung, Variablen und Zuweisungen	113
2.3.1	Primitiv- oder Verweistyp	114
2.3.2	Primitive Datentypen im Überblick	115
2.3.3	Variablendeklarationen	118
2.3.4	Automatisches Feststellen der Typen mit var	120
2.3.5	Finale Variablen und der Modifizierer final	121
2.3.6	Unbenannte Variablen mit _	122
2.3.7	Konsoleneingaben	123
2.3.8	Wahrheitswerte	125
2.3.9	Ganzzahlige Datentypen	125
2.3.10	Unterstriche in Zahlen	127
2.3.11	Alphanumerische Zeichen	128
2.3.12	Gleitkommazahlen mit den Datentypen float und double	129
2.3.13	Gute Namen, schlechte Namen	130
2.3.14	Keine automatische Initialisierung von lokalen Variablen	131
2.4	Ausdrücke, Operanden und Operatoren	132
2.4.1	Die Arten von Operatoren	132
2.4.2	Zuweisungsoperator	133
2.4.3	Arithmetische Operatoren	134
2.4.4	Unäres Minus und Plus	138
2.4.5	Präfix- oder Postfix-Inkrement und -Dekrement	139
2.4.6	Zuweisung mit Operation (Verbundoperator)	141
2.4.7	Die relationalen Operatoren und die Gleichheitsoperatoren	142
2.4.8	Logische Operatoren: Nicht, Und, Oder, XOR	143
2.4.9	Kurzschluss-Operatoren	144
2.4.10	Der Rang der Operatoren in der Auswertungsreihenfolge	146
2.4.11	Die Typumwandlung (das Casting)	149
2.4.12	Überladenes Plus für Strings	154
2.4.13	Operator vermisst *	155
2.5	Bedingte Anweisungen oder Fallunterscheidungen	156
2.5.1	Verzweigung mit der if-Anweisung	156
2.5.2	Die Alternative mit einer if-else-Anweisung wählen	159
2.5.3	Der Bedingungsoperator	162
2.5.4	Die switch-Anweisung bietet die Alternative	165
2.5.5	Switch-Ausdrücke	172

2.6	Immer das Gleiche mit den Schleifen	175
2.6.1	Die while-Schleife	176
2.6.2	Die do-while-Schleife	177
2.6.3	Die for-Schleife	179
2.6.4	Schleifenbedingungen und Vergleiche mit == *	184
2.6.5	Schleifenabbruch mit break und zurück zum Test mit continue	186
2.6.6	break und continue mit Marken *	189
2.7	Methoden deklarieren	193
2.7.1	Bestandteile einer Methode	193
2.7.2	Methoden ohne Parameter deklarieren	195
2.7.3	Parameter, Argument und Wertübergabe	196
2.7.4	Signatur-Beschreibung in der Java-API	199
2.7.5	Methoden vorzeitig mit return beenden	201
2.7.6	Methoden mit Rückgaben	201
2.7.7	Methoden überladen	203
2.7.8	Vorgegebener Wert für nicht aufgeführte Argumente	205
2.7.9	Rückgabepfade und Kontrollfluss in Methoden	206
2.7.10	Rekursive Methoden *	210
2.8	Methoden brauchen ein Zuhause: Instanz oder Klasse	214
2.8.1	Statische Methoden (Klassenmethoden)	214
2.8.2	Kompakte Quelldatei und »echte« Klassen	215
2.8.3	Statische Methoden von anderen Klassen aufrufen	217
2.8.4	Gültigkeitsbereich	218
2.9	Zum Weiterlesen	220

3 Klassen und Objekte 221

3.1	Objektorientierte Programmierung (OOP)	221
3.1.1	Warum überhaupt OOP?	222
3.1.2	Denk ich an Java, denk ich an Wiederverwendbarkeit	222
3.2	Eigenschaften einer Klasse	223
3.2.1	Klassenarbeit mit Point	224
3.3	Natürlich modellieren mit der UML (Unified Modeling Language) *	225
3.3.1	Wichtige Diagrammtypen der UML *	225
3.4	Neue Objekte erzeugen	227
3.4.1	Ein Objekt einer Klasse mit dem Schlüsselwort new anlegen	227
3.4.2	Deklarieren von Referenzvariablen	227

3.4.3	Jetzt mach mal 'nen Punkt: Zugriff auf Objektvariablen und -methoden	228
3.4.4	Überblick über Point-Methoden	232
3.4.5	Konstruktoren nutzen	236
3.5	ZZZZZnake	237
3.5.1	Erweiterung	239
3.6	Pakete schnüren, Importe und Compilationseinheiten	239
3.6.1	Java-Pakete	239
3.6.2	Pakete der Standardbibliothek	240
3.6.3	Volle Qualifizierung und import-Deklaration	240
3.6.4	Mit import p1.p2.* alle Typen eines Pakets erreichen	242
3.6.5	Modul-Import	243
3.6.6	Hierarchische Strukturen über Pakete und die Spiegelung im Dateisystem	246
3.6.7	Die package-Deklaration	246
3.6.8	Unbenanntes Paket (default package)	248
3.6.9	Compilationseinheit (Compilation Unit)	248
3.6.10	Statischer Import *	249
3.7	Mit Referenzen arbeiten, Vielfalt, Identität, Gleichwertigkeit	250
3.7.1	null-Referenz und die Frage der Philosophie	250
3.7.2	Alles auf null? Referenzen testen	252
3.7.3	Zuweisungen bei Referenzen	253
3.7.4	Methoden mit Referenztypen als Parameter	255
3.7.5	Identität von Objekten	259
3.7.6	Gleichwertigkeit und die Methode equals(...)	259
3.7.7	Ausblick: Value Types – Objekte ohne Identität	261
3.8	Der Zusammenhang von new, Heap und Garbage-Collector	262
3.8.1	Heap-Speicher	262
3.8.2	Automatische Speicherbereinigung/Garbage-Collector (GC) – es ist dann mal weg	263
3.8.3	OutOfMemoryError	263
3.9	Zum Weiterlesen	264
4	Arrays und ihre Anwendungen	265
4.1	Einfache Feldarbeit	265
4.1.1	Grundbestandteile	266
4.1.2	Deklaration von Array-Variablen	267
4.1.3	Array-Objekte mit new erzeugen	268

4.1.4	Arrays mit { Inhalt }	269
4.1.5	Die Länge eines Arrays über die Objektvariable length auslesen	270
4.1.6	Zugriff auf die Elemente über den Index	270
4.1.7	Typische Array-Fehler	272
4.1.8	Arrays an Methoden übergeben	274
4.1.9	Mehrere Rückgabewerte *	274
4.1.10	Vorinitialisierte Arrays	275
4.2	Die erweiterte for-Schleife	277
4.2.1	Anonyme Arrays in der erweiterten for-Schleife nutzen	278
4.2.2	Umsetzung und Einschränkung	278
4.2.3	Beispiel: Arrays mit Strings durchsuchen	279
4.2.4	Zufällige Spielerpositionen erzeugen	279
4.3	Methode mit variabler Argumentanzahl (Varargs)	281
4.3.1	System.out.printf(...) nimmt eine beliebige Anzahl von Argumenten an	281
4.3.2	Durchschnitt finden von variablen Argumenten	282
4.3.3	Varargs-Designtipps	283
4.4	Mehrdimensionale Arrays *	284
4.4.1	Mehrdimensionale Array-Objekte mit new aufbauen	284
4.4.2	Anlegen und Initialisieren in einem Schritt	284
4.4.3	Zugriff auf Elemente	285
4.4.4	length bei mehrdimensionalen Arrays	285
4.4.5	Zweidimensionale Arrays mit ineinander verschachtelten Schleifen ablaufen	285
4.4.6	Nichtrechteckige Arrays *	287
4.5	Bibliotheksunterstützung von Arrays	290
4.5.1	Klonen kann sich lohnen – Arrays vermehren	290
4.5.2	Warum »können« Arrays so wenig?	291
4.5.3	Array-Inhalte kopieren	291
4.6	Die Klasse Arrays zum Vergleichen, Füllen, Suchen und Sortieren nutzen	292
4.6.1	String-Repräsentation eines Arrays	293
4.6.2	Sortieren	293
4.6.3	Arrays von Primitiven mit Arrays.equals(...) und Arrays.deepEquals(...) vergleichen *	295
4.6.4	Objekt-Arrays mit Arrays.equals(...) und Arrays.deepEquals(...) vergleichen *	296
4.6.5	Unterschiede suchen mit mismatch (...) *	297
4.6.6	Füllen von Arrays *	298
4.6.7	Array-Abschnitte kopieren *	299
4.6.8	Halbierungssuche *	300

4.6.9	Array-Vergleiche mit <code>compare(...)</code> und <code>compareUnsigned(...)</code>	302
4.6.10	Arrays zu Listen mit <code>Arrays.asList(...)</code> – praktisch für die Suche und zum Vergleichen *	302
4.6.11	Eine lange Schlange	304
4.7	Der Startpunkt für das Laufzeitsystem: <code>main(...)</code>	307
4.7.1	Kommandozeilenargumente	307
4.7.2	Deklaration der Startmethode mit Array für Argumente	307
4.7.3	Kommandozeilenargumente verarbeiten	308
4.7.4	Der Rückgabotyp von <code>main(...)</code> und <code>System.exit(int)</code> *	309
4.8	Zum Weiterlesen	310

5 Der Umgang mit Zeichen und Zeichenketten 311

5.1	Von ASCII über ISO-8859-1 zu Unicode	311
5.1.1	ASCII	311
5.1.2	ISO/IEC 8859-1	312
5.1.3	Unicode	313
5.1.4	Unicode-Zeichenkodierung	314
5.1.5	Escape-Sequenzen/Fluchtsymbole	315
5.1.6	Schreibweise für Unicode-Zeichen und Unicode-Escapes	316
5.1.7	Java-Versionen gehen mit dem Unicode-Standard Hand in Hand *	318
5.2	Datentypen für Zeichen und Zeichenketten	319
5.3	Die Character-Klasse	320
5.3.1	Ist das so?	320
5.3.2	Zeichen in Großbuchstaben/Kleinbuchstaben konvertieren	322
5.3.3	Vom Zeichen zum String	323
5.3.4	Von char in int: vom Zeichen zur Zahl *	323
5.4	Datentypen für Zeichenketten	325
5.4.1	Die Klasse String	326
5.4.2	Die Klassen <code>StringBuilder</code> / <code>StringBuffer</code>	326
5.4.3	Der Basistyp <code>CharSequence</code> für Zeichenketten	327
5.4.4	Enthält ein String ein char-Array? *	327
5.5	Die Klasse String und ihre Methoden	327
5.5.1	String-Literale als String-Objekte für konstante Zeichenketten	328
5.5.2	Konkatenation mit +	328
5.5.3	Mehrzeilige Textblöcke mit <code>"""</code>	328

5.5.4	String-Länge und Test auf Leer-String	333
5.5.5	Zugriff auf ein bestimmtes Zeichen mit charAt(int)	335
5.5.6	Nach enthaltenen Zeichen und Zeichenketten suchen	336
5.5.7	Das Hangman-Spiel	339
5.5.8	Gut, dass wir verglichen haben	341
5.5.9	String-Teile extrahieren	345
5.5.10	Strings anhängen, zusammenfügen, Groß-/Kleinschreibung und Weißraum	350
5.5.11	Gesucht, gefunden, ersetzt	353
5.5.12	String-Objekte mit Konstruktoren und aus Wiederholungen erzeugen *	355
5.6	Veränderbare Zeichenketten mit StringBuilder	359
5.6.1	Überblick	359
5.6.2	Anlegen von StringBuilder-Objekten	359
5.6.3	StringBuilder in andere Zeichenkettenformate konvertieren	360
5.6.4	Zeichen(folgen) erfragen	360
5.6.5	Daten anhängen	360
5.6.6	Zeichen(folgen) setzen, löschen und umdrehen	362
5.6.7	Länge und Kapazität eines StringBuilder-Objekts *	364
5.6.8	Vergleich von StringBuilder-Instanzen und Strings mit StringBuilder	365
5.6.9	hashCode() bei StringBuilder *	367
5.7	CharSequence als Basistyp	368
5.7.1	Basisoperationen der Schnittstelle	369
5.7.2	Statische compare(...) -Methode in CharSequence	370
5.7.3	Default-Methoden in der Schnittstelle CharSequence *	370
5.8	Konvertieren zwischen Primitiven und Strings	371
5.8.1	Unterschiedliche Typen in String-Repräsentationen konvertieren	371
5.8.2	String-Inhalt in einen primitiven Wert konvertieren	373
5.8.3	String-Repräsentation in den Formaten Binär, Hex und Oktal *	375
5.9	Strings verketten (konkatenieren)	379
5.9.1	Strings mit StringJoiner verketten	379
5.10	Zerlegen von Zeichenketten	381
5.10.1	Splitten von Zeichenketten mit split(...)	382
5.10.2	Yes we can, yes we scan – die Klasse Scanner	383
5.11	Ausgaben formatieren	386
5.11.1	Formatieren und Ausgeben mit format(...)	386
5.12	Zum Weiterlesen	393

6	Eigene Klassen schreiben	395
6.1	Eigene Klassen mit Eigenschaften deklarieren	395
6.1.1	Minimalklasse	396
6.1.2	Objektvariablen deklarieren	396
6.1.3	Methoden deklarieren	399
6.1.4	Verdeckte (shadowed) Variablen	402
6.1.5	Die this-Referenz	403
6.2	Privatsphäre und Sichtbarkeit	407
6.2.1	Für die Öffentlichkeit: public	408
6.2.2	Kein Public Viewing – Passwörter sind privat	408
6.2.3	Wieso nicht freie Methoden und Variablen für alle?	410
6.2.4	Privat ist nicht ganz privat: Es kommt darauf an, wer's sieht *	410
6.2.5	Zugriffsmethoden für Objektvariablen deklarieren	411
6.2.6	Setter und Getter nach der JavaBeans-Spezifikation	412
6.2.7	Paketsichtbar	414
6.2.8	Zusammenfassung zur Sichtbarkeit	415
6.3	Eine für alle – statische Methoden und Klassenvariablen	418
6.3.1	Warum statische Eigenschaften sinnvoll sind	419
6.3.2	Statische Eigenschaften mit static	419
6.3.3	Statische Eigenschaften über Referenzen nutzen? *	422
6.3.4	Warum die Groß- und Kleinschreibung wichtig ist *	423
6.3.5	Statische Variablen zum Datenaustausch *	424
6.3.6	Statische Eigenschaften und Objekteigenschaften *	425
6.4	Konstanten und Aufzählungen	426
6.4.1	Konstanten über statische finale Variablen	426
6.4.2	Typunsichere Aufzählungen	428
6.4.3	Aufzählungstypen: typsichere Aufzählungen mit enum	430
6.5	Objekte anlegen und zerstören	435
6.5.1	Konstruktoren schreiben	435
6.5.2	Verwandtschaft von Methode und Konstruktor	437
6.5.3	Der Standard-Konstruktor (default constructor)	437
6.5.4	Parametrisierte und überladene Konstruktoren	438
6.5.5	Copy-Konstruktor	441
6.5.6	Einen anderen Konstruktor der gleichen Klasse mit this(...) aufrufen	442
6.5.7	Immutable-Objekte und With-Methoden	445
6.5.8	Ihr fehlt uns nicht – der Garbage-Collector	447

6.6	Klassen- und Objektinitialisierung *	449
6.6.1	Initialisierung von Objektvariablen	449
6.6.2	Statische Blöcke als Klasseninitialisierer	451
6.6.3	Initialisierung von Klassenvariablen	452
6.6.4	Eincompilierte Belegungen der finalen Klassenvariablen	453
6.6.5	Exemplarinitialisierer (Instanzinitialisierer)	454
6.6.6	Finale Werte im Konstruktor und in statischen Blöcken setzen	457
6.7	Zum Weiterlesen	458
7	Objektorientierte Beziehungsfragen	459
7.1	Assoziationen zwischen Objekten	459
7.1.1	Assoziationstypen	459
7.1.2	Unidirektionale 1:1-Beziehung	460
7.1.3	Zwei Freunde müsst ihr werden – bidirektionale 1:1-Beziehungen	461
7.1.4	Unidirektionale 1:n-Beziehung	463
7.2	Vererbung	470
7.2.1	Vererbung in Java	471
7.2.2	Ereignisse modellieren	471
7.2.3	Die implizite Basisklasse java.lang.Object	473
7.2.4	Einfach- und Mehrfachvererbung *	474
7.2.5	Sehen Kinder alles? Die Sichtbarkeit protected	475
7.2.6	Konstruktoren in der Vererbung und super(...)	476
7.3	Typen in Hierarchien	481
7.3.1	Automatische und explizite Typumwandlung	481
7.3.2	Das Substitutionsprinzip	484
7.3.3	Typen mit dem instanceof-Operator testen	487
7.3.4	Pattern-Matching bei instanceof	489
7.3.5	Pattern-Matching bei switch	491
7.4	Methoden überschreiben	494
7.4.1	Methoden in Unterklassen mit neuem Verhalten ausstatten	494
7.4.2	Mit super an die Eltern	498
7.5	Drum prüfe, wer sich dynamisch bindet	501
7.5.1	Gebunden an toString()	501
7.5.2	Implementierung von System.out.println(Object)	503

7.6	Finale Klassen und finale Methoden	504
7.6.1	Finale Klassen	504
7.6.2	Nicht überschreibbare (finale) Methoden	504
7.7	Abstrakte Klassen und abstrakte Methoden	506
7.7.1	Abstrakte Klassen	506
7.7.2	Abstrakte Methoden	508
7.8	Weiteres zum Überschreiben und dynamischen Binden	514
7.8.1	Nicht dynamisch gebunden bei privaten, statischen und finalen Methoden	515
7.8.2	Kovariante Rückgabetypen	515
7.8.3	Array-Typen und Kovarianz *	516
7.8.4	Dynamisch gebunden auch bei Konstruktoraufrufen *	517
7.8.5	Keine dynamische Bindung bei überdeckten Objektvariablen *	519
7.9	Zum Weiterlesen und Programmieraufgabe	520

8 Records, Schnittstellen, Aufzählungen, versiegelte Klassen 523

8.1	Records	523
8.1.1	Einfache Records	523
8.1.2	Records mit Methoden	525
8.1.3	Konstruktoren von Records anpassen	527
8.1.4	Konstruktoren ergänzen	529
8.1.5	Record-Patterns zur Destrukturierung	530
8.1.6	Records: Zusammenfassung	534
8.2	Schnittstellen	535
8.2.1	Schnittstellen sind neue Typen	535
8.2.2	Schnittstellen deklarieren	535
8.2.3	Abstrakte Methoden in Schnittstellen	536
8.2.4	Schnittstellen implementieren	537
8.2.5	Ein Polymorphie-Beispiel mit Schnittstellen	540
8.2.6	Klassen können eine Oberklasse haben und Schnittstellen implementieren	543
8.2.7	Records implementieren Schnittstellen	546
8.2.8	Die Mehrfachvererbung bei Schnittstellen	547
8.2.9	Erweitern von Interfaces – Unterschnittstellen	550
8.2.10	Geldkraken im Direktvergleich – mit der Schnittstelle Comparable	554

8.2.11	Konstantendeklarationen bei Schnittstellen	556
8.2.12	Nachträgliches Implementieren von Schnittstellen *	557
8.2.13	Statische ausprogrammierte Methoden in Schnittstellen	558
8.2.14	Erweitern und Ändern von Schnittstellen	559
8.2.15	Default-Methoden	561
8.2.16	Öffentliche und private Schnittstellenmethoden	567
8.2.17	Erweiterte Schnittstellen, Mehrfachvererbung und Mehrdeutigkeiten *	567
8.2.18	Bausteine bilden mit Default-Methoden *	572
8.2.19	Markierungsschnittstellen *	575
8.2.20	(Abstrakte) Klassen und Schnittstellen im Vergleich	575
8.3	Aufzählungstypen	576
8.3.1	Existierende Methoden auf Enum-Objekten	577
8.3.2	Aufzählungen mit eigenen Methoden *	580
8.3.3	enum mit eigenen Konstruktoren und Initialisierern	583
8.3.4	enum kann Schnittstellen implementieren	586
8.4	Versiegelte Klassen und Schnittstellen	587
8.4.1	Grenzen von Aufzählungstypen und regulären Klassen	588
8.4.2	Versiegelte Klassen und Schnittstellen (sealed classes/interfaces)	589
8.4.3	Unterklassen sind final, sealed, non-sealed	591
8.4.4	Vollständige Abdeckung von Aufzählungen	592
8.4.5	Abkürzende Schreibweisen	593
8.4.6	Versiegelte Schnittstellen und Records	594
8.5	Zum Weiterlesen	595
9	Ausnahmen müssen sein	597
9.1	Problemgebiete einzäunen	598
9.1.1	Exceptions in Java mit try und catch	598
9.1.2	Geprüfte und ungeprüfte Ausnahmen	599
9.1.3	Eine NumberFormatException fliegt (ungeprüfte Ausnahme)	599
9.1.4	Datum-Zeitstempel in Textdatei anhängen (geprüfte Ausnahme)	601
9.1.5	Wiederholung abgebrochener Bereiche *	604
9.1.6	Bitte nicht schlucken – leere catch-Blöcke	605
9.1.7	Mehrere Ausnahmen auffangen	605
9.1.8	Zusammenfassen gleicher catch-Blöcke mit dem multi-catch	606
9.2	Ausnahmen mit throws nach oben reichen	607
9.2.1	throws bei Konstruktoren und Methoden	607

9.3	Die Klassenhierarchie der Ausnahmen	608
9.3.1	Eigenschaften des Ausnahmeobjekts	608
9.3.2	Basistyp Throwable	610
9.3.3	Die Exception-Hierarchie	610
9.3.4	Oberausnahmen auffangen	611
9.3.5	Schon gefangen?	613
9.3.6	Ablauf einer Ausnahmesituation	614
9.3.7	Nicht zu allgemein fangen!	614
9.3.8	Bekannte RuntimeException-Klassen	616
9.3.9	Kann man abfangen, muss man aber nicht	617
9.4	Abschlussbehandlung mit finally	618
9.4.1	Die ignorante Version	618
9.4.2	Der gut gemeinte Versuch	619
9.4.3	Ab jetzt wird scharf geschlossen	620
9.4.4	Zusammenfassung	621
9.4.5	Ein try ohne catch, aber ein try-finally	622
9.5	Auslösen eigener Exceptions	623
9.5.1	Mit throw Ausnahmen auslösen	623
9.5.2	Vorhandene ungeprüfte Ausnahmetypen kennen und nutzen	625
9.5.3	Parameter testen und gute Fehlermeldungen	628
9.5.4	Neue Exception-Klassen deklarieren	630
9.5.5	Eigene Ausnahmen als Unterklassen von Exception oder RuntimeException?	631
9.5.6	Ausnahmen abfangen und weiterleiten *	634
9.5.7	Aufruf-Stack von Ausnahmen verändern *	635
9.5.8	Geschachtelte Ausnahmen *	636
9.6	try mit Ressourcen (automatisches Ressourcenmanagement)	640
9.6.1	Ressourcenmanagement mit try mit Ressourcen	640
9.6.2	Die Schnittstelle AutoCloseable	641
9.6.3	Ausnahmen vom close()	642
9.6.4	Typen, die AutoCloseable und Closeable sind	643
9.6.5	Mehrere Ressourcen nutzen	645
9.6.6	try mit Ressourcen auf null-Ressourcen	646
9.6.7	Unterdrückte Ausnahmen *	646
9.7	Besonderheiten bei der Ausnahmebehandlung *	650
9.7.1	Rückgabewerte bei ausgelösten Ausnahmen	650
9.7.2	Ausnahmen und Rückgaben verschwinden – das Duo return und finally	650
9.7.3	throws bei überschriebenen Methoden	652
9.7.4	Nicht erreichbare catch-Klauseln	654

9.8	Harte Fehler – Error *	655
9.9	Zusicherungen im Programmcode (Assertions) *	656
9.9.1	Der assert-Anweisung	656
9.9.2	Assertions aktivieren	657
9.9.3	Feinsteuerung von Assertions	658
9.10	Zum Weiterlesen	659

10 Geschachtelte Typen

10.1	Geschachtelte Klassen, Records, Schnittstellen und Aufzählungen	661
10.2	Statische geschachtelte Typen	663
10.2.1	Modifizierer und Sichtbarkeit	664
10.2.2	Records als Behälter	664
10.2.3	Umsetzung von statischen geschachtelten Typen *	664
10.3	Nichtstatische geschachtelte Typen	665
10.3.1	Eine Schokoladenfabrik mit Schokoladenpresse	665
10.3.2	Exemplare innerer Klassen erzeugen	666
10.3.3	Die this-Referenz	666
10.3.4	Vom Compiler generierte Klassendateien *	668
10.4	Lokale Klassen	668
10.4.1	Beispiel mit eigener Klassendeklaration	668
10.4.2	Lokale Klasse für einen Timer nutzen	669
10.5	Anonyme innere Klassen	670
10.5.1	Nutzung einer anonymen inneren Klasse für den Timer	671
10.5.2	Umsetzung innerer anonymer Klassen *	672
10.5.3	Konstrukturen innerer anonymer Klassen	672
10.6	Vermischtes	675
10.6.1	Zugriff auf lokale Variablen aus lokalen und anonymen Klassen *	675
10.6.2	this in Unterklassen *	675
10.6.3	Geschachtelte Klassen greifen auf private Eigenschaften zu	676
10.6.4	Nester	678
10.7	Zum Weiterlesen	679

11	Besondere Typen der Java SE	681
11.1	Object ist die Mutter aller Klassen	682
11.1.1	Klassenobjekte	682
11.1.2	Objektidentifikation mit toString()	683
11.1.3	Objektgleichwertigkeit mit equals(...) und Identität	685
11.1.4	Klonen eines Objekts mit clone() *	691
11.1.5	Hashwerte über hashCode() liefern *	696
11.1.6	System.identityHashCode(...) und das Problem der nicht eindeutigen Objektverweise *	702
11.1.7	Synchronisation *	703
11.2	Schwache Referenzen und Cleaner	704
11.3	Die Utility-Klasse java.util.Objects	705
11.3.1	Eingebaute null-Tests für equals()/hashCode()	705
11.3.2	Objects.toString(...)	706
11.3.3	null-Prüfungen mit eingebauter Ausnahmebehandlung	707
11.3.4	Tests auf null	708
11.3.5	Indexbezogene Programmargumente auf Korrektheit prüfen	708
11.4	Vergleichen von Objekten und Ordnung herstellen	709
11.4.1	Natürlich geordnet oder nicht?	709
11.4.2	compare*()-Methode der Schnittstellen Comparable und Comparator	711
11.4.3	Rückgabewerte kodieren die Ordnung	711
11.4.4	Mit einem Beispiel-Comparator Süßigkeiten nach Kalorien sortieren	712
11.4.5	Tipps für Comparator- und Comparable-Implementierungen	714
11.4.6	Statische und Default-Methoden in Comparator	715
11.5	Wrapper-Klassen und Autoboxing	718
11.5.1	Wrapper-Objekte erzeugen	720
11.5.2	Konvertierungen in eine String-Repräsentation	721
11.5.3	Von einer String-Repräsentation parsen	722
11.5.4	Die Basisklasse Number für numerische Wrapper-Objekte	723
11.5.5	Vergleiche durchführen mit compare*(...), compareTo(...), equals(...) und Hashwerten	724
11.5.6	Statische Reduzierungsmethoden in Wrapper-Klassen	727
11.5.7	Konstanten für die Größe eines primitiven Typs	728
11.5.8	Behandeln von vorzeichenlosen Zahlen *	729
11.5.9	Die Klassen Integer und Long	730
11.5.10	Die Klassen Double und Float für Gleitkommazahlen	732
11.5.11	Die Boolean-Klasse	732
11.5.12	Autoboxing: Boxing und Unboxing	733

11.6	Iterator, Iterable *	737
11.6.1	Die Schnittstelle Iterator	738
11.6.2	Wer den Iterator liefert	741
11.6.3	Die Schnittstelle Iterable	741
11.6.4	Erweitertes for und Iterable	742
11.6.5	Interne Iteration	742
11.6.6	Ein eigenes Iterable implementieren *	743
11.7	Annotationen in der Java SE	745
11.7.1	Orte für Annotationen	745
11.7.2	Annotationstypen aus java.lang	746
11.7.3	@Deprecated	747
11.7.4	Annotationen mit zusätzlichen Informationen	747
11.7.5	@SuppressWarnings	748
11.8	Zum Weiterlesen	750

12 Generics<T>

12.1	Einführung in Java Generics	751
12.1.1	Mensch versus Maschine – Typprüfung des Compilers und der Laufzeitumgebung	751
12.1.2	Raketen	752
12.1.3	Generische Typen deklarieren	754
12.1.4	Generics nutzen	756
12.1.5	Diamonds are forever	759
12.1.6	Generische Schnittstellen	762
12.1.7	Generische Methoden/Konstruktoren und Typ-Inferenz	764
12.2	Umsetzen der Generics, Typlöschung und Raw-Types	768
12.2.1	Realisierungsmöglichkeiten	768
12.2.2	Typlöschung (Type Erasure)	768
12.2.3	Raw-Type	769
12.2.4	Probleme der Typlöschung	772
12.3	Die Typen über Bounds einschränken	776
12.3.1	Einfache Einschränkungen mit extends	777
12.3.2	Weitere Obertypen mit &	780
12.4	Typparameter in der throws-Klausel *	780
12.4.1	Deklaration einer Klasse mit Typvariable <E extends Exception>	780
12.4.2	Parametrisierter Typ bei Typvariable <E extends Exception>	781

12.5 Generics und Vererbung, Invarianz	784
12.5.1 Arrays sind kovariant	784
12.5.2 Generics sind nicht kovariant, sondern invariant	784
12.5.3 Wildcards mit ?	785
12.5.4 Bounded Wildcards	788
12.5.5 Bounded-Wildcard-Typen und Bounded-Typvariablen	791
12.5.6 Das LESS-Prinzip	794
12.6 Konsequenzen der Typlöschung: Typ-Token, Arrays und Brücken *	796
12.6.1 Typ-Token	796
12.6.2 Super-Type-Token	798
12.6.3 Generics und Arrays	799
12.6.4 Brückenmethoden *	800
12.7 Zum Weiterlesen	806

13 Lambda-Ausdrücke und funktionale Programmierung

807

13.1 Funktionale Schnittstellen und Lambda-Ausdrücke	807
13.1.1 Klassen implementieren Schnittstellen	807
13.1.2 Lambda-Ausdrücke implementieren Schnittstellen	809
13.1.3 Funktionale Schnittstellen	810
13.1.4 Der Typ eines Lambda-Ausdrucks ergibt sich durch den Zieltyp	811
13.1.5 Annotation @FunctionalInterface	816
13.1.6 Syntax für Lambda-Ausdrücke	817
13.1.7 Die Umgebung der Lambda-Ausdrücke und Variablenzugriffe	821
13.1.8 Ausnahmen in Lambda-Ausdrücken	827
13.1.9 Klassen mit einer abstrakten Methode als funktionale Schnittstelle? *	831
13.2 Methodenreferenz	831
13.2.1 Motivation	832
13.2.2 Methodenreferenzen mit ::	832
13.2.3 Varianten von Methodenreferenzen	833
13.3 Konstruktorreferenz	836
13.3.1 Konstruktorreferenzen schreiben	837
13.3.2 Parameterlose und parametrisierte Konstruktoren	838
13.3.3 Nützliche vordefinierte Schnittstellen für Konstruktorreferenzen	839
13.4 Funktionale Programmierung	839
13.4.1 Code = Daten	840

13.4.2	Programmierparadigmen: imperativ oder deklarativ	841
13.4.3	Das Wesen der funktionalen Programmierung	842
13.4.4	Funktionale Programmierung und funktionale Programmiersprachen	844
13.4.5	Funktionen höherer Ordnung am Beispiel von Comparator	846
13.4.6	Lambda-Ausdrücke als Abbildungen bzw. Funktionen betrachten	847
13.5	Funktionale Schnittstellen aus dem java.util.function-Paket	848
13.5.1	Blöcke mit Code und die funktionale Schnittstelle Consumer	849
13.5.2	Supplier	851
13.5.3	Prädikate und java.util.function.Predicate	851
13.5.4	Funktionen über die funktionale Schnittstelle java.util.function.Function	854
13.5.5	Ein bisschen Bi	858
13.5.6	Funktionale Schnittstellen mit Primitiven	861
13.6	Optional ist keine Nullnummer	863
13.6.1	Einsatz von null	864
13.6.2	Der Optional-Typ	866
13.6.3	Erst mal funktional mit Optional	868
13.6.4	Primitiv-Optionales mit speziellen Optional*-Klassen	871
13.7	Was ist jetzt so funktional?	874
13.7.1	Wiederverwertbarkeit	874
13.7.2	Zustandslos, immutable	875
13.8	Zum Weiterlesen	876

14 Architektur, Design und angewandte Objektorientierung 877

14.1	SOLIDe Modellierung	877
14.1.1	DRY, KISS und YAGNI	878
14.1.2	SOLID	878
14.1.3	Sei nicht STUPID	880
14.2	Architektur, Design und Implementierung	881
14.2.1	Implementierung und Idiome	881
14.2.2	Design	881
14.2.3	Architektur	882
14.3	Design-Patterns (Entwurfsmuster)	882
14.3.1	Motivation für Design-Patterns	883

14.3.2	Singleton	883
14.3.3	Fabrikmethoden	885
14.3.4	Das Beobachter-Pattern mit Listener realisieren	886
14.4	Zum Weiterlesen	890

15 Java Platform Module System 891

15.1	Klassenlader (Class Loader) und Modul-/Klassenpfad	891
15.1.1	Klassenladen auf Abruf	891
15.1.2	Klassenlader bei der Arbeit zusehen	892
15.1.3	JMOD-Dateien und JAR-Dateien	893
15.1.4	Woher die Klassen kommen: Suchorte und spezielle Klassenlader	894
15.1.5	Setzen des Suchpfads	895
15.2	Module im JPMS einbinden	897
15.2.1	Wer sieht wen?	898
15.2.2	Interne Plattformeigenschaften nutzen, --add-exports	899
15.2.3	Neue Module einbinden	901
15.3	Eigene Module entwickeln	902
15.3.1	Modul com.tutego.candytester	902
15.3.2	Moduldeklaration mit module-info.java und exports	903
15.3.3	Modul com.tutego.main	904
15.3.4	Modulinfodatei mit requires	904
15.3.5	Module mit den JVM-Optionen -p und -m ausführen	905
15.3.6	Modulinfodatei-Experimente	906
15.3.7	Automatische Module	907
15.3.8	Unbenanntes Modul	908
15.3.9	Lesbarkeit und Zugreifbarkeit	908
15.3.10	Modul-Migration	909
15.4	Zum Weiterlesen	911

16 Die Klassenbibliothek 913

16.1	Die Java-Klassenphilosophie	913
16.1.1	Modul, Paket, Typ	913
16.1.2	Übersicht über die Pakete der Standardbibliothek	915

16.2 Einfache Zeitmessung und Profiling *	917
16.2.1 Profiler *	918
16.3 Die Klasse Class	918
16.3.1 An ein Class-Objekt kommen	918
16.3.2 Eine Class ist ein Type	921
16.4 Die Utility-Klassen System und Properties	921
16.4.1 Speicher der JVM	922
16.4.2 Systemeigenschaften der Java-Umgebung	923
16.4.3 Eigene Properties von der Konsole aus setzen *	925
16.4.4 Zeilenumbruchzeichen, line.separator	927
16.4.5 Umgebungsvariablen des Betriebssystems	927
16.5 Sprachen der Länder	929
16.5.1 Sprachen in Regionen über Locale-Objekte	929
16.6 Wichtige Datum-Klassen im Überblick	933
16.6.1 Der 1.1.1970	934
16.6.2 System.currentTimeMillis()	934
16.6.3 Einfache Zeitumrechnungen durch TimeUnit	934
16.7 Date-Time-API	935
16.7.1 Erster Überblick	936
16.7.2 Menschenzeit und Maschinenzeit	937
16.7.3 Die Datumsklasse LocalDate	939
16.8 Logging mit Java	940
16.8.1 Logging-APIs	941
16.8.2 Logging mit java.util.logging	941
16.9 Maven: Build-Management und Abhängigkeiten auflösen	943
16.9.1 Dependency hinzunehmen	944
16.9.2 Lokales und das Remote Repository	945
16.9.3 Lebenszyklus, Phasen und Maven-Plugins	945
16.10 Zum Weiterlesen	945

17 Einführung in die nebenläufige Programmierung 947

17.1 Nebenläufigkeit und Parallelität	947
17.1.1 Multitasking, Prozesse und Threads	948
17.1.2 Wie nebenläufige Programme die Geschwindigkeit steigern können	949
17.1.3 Java-Threads	950
17.1.4 Was Java für Nebenläufigkeit alles bietet	951

17.2	Existierende Threads und neue Threads erzeugen	951
17.2.1	Main-Thread	951
17.2.2	Wer bin ich?	952
17.2.3	Die Schnittstelle Runnable implementieren	952
17.2.4	Thread aufbauen	953
17.2.5	Thread mit Runnable starten	954
17.2.6	Runnable parametrisieren	956
17.2.7	Schläfer gesucht	957
17.2.8	Wann Threads fertig sind	958
17.2.9	Einen Thread höflich mit Interrupt beenden	959
17.2.10	Unbehandelte Ausnahmen, Thread-Ende und UncaughtExceptionHandler	961
17.2.11	Ein Rendezvous mit join(...) *	963
17.3	Der Ausführer (Executor) kommt ins Spiel	965
17.3.1	Die Schnittstelle Executor	966
17.3.2	Glücklich in der Gruppe – die Thread-Pools	967
17.3.3	Threads mit Rückgabe über Callable	970
17.3.4	Erinnerungen an die Zukunft – die Future-Rückgabe	972
17.3.5	Mehrere Callable-Objekte abarbeiten	975
17.3.6	CompletionService und ExecutorCompletionService	976
17.3.7	ScheduledExecutorService: wiederholende Aufgaben und Zeitsteuerungen	978
17.3.8	Asynchrones Programmieren mit CompletableFuture (CompletionStage) ...	978
17.4	Zum Weiterlesen	981

18 Einführung in Datenstrukturen und Algorithmen 983

18.1	Listen	983
18.1.1	Erstes Listen-Beispiel	984
18.1.2	Auswahlkriterium ArrayList oder LinkedList	985
18.1.3	Die Schnittstelle List	986
18.1.4	ArrayList	992
18.1.5	LinkedList	993
18.1.6	Der Array-Adapter Arrays.asList(...)	994
18.1.7	toArray(...) von Collection verstehen – die Gefahr einer Falle erkennen	996
18.1.8	Primitive Elemente in Datenstrukturen verwalten	998

18.2 Mengen (Sets)	998
18.2.1 Ein erstes Mengen-Beispiel	999
18.2.2 Methoden der Schnittstelle Set	1001
18.2.3 HashSet	1003
18.2.4 TreeSet – die sortierte Menge	1004
18.2.5 Die Schnittstellen NavigableSet und SortedSet	1006
18.2.6 LinkedHashSet	1008
18.3 Assoziative Speicher	1009
18.3.1 Die Klassen HashMap und TreeMap, statische Map-Methoden	1010
18.3.2 Einfügen und Abfragen des Assoziativspeichers	1013
18.4 Ausgewählte Basistypen	1015
18.5 Java-Stream-API	1016
18.5.1 Deklaratives Programmieren	1016
18.5.2 Interne versus externe Iteration	1017
18.5.3 Was ist ein Stream?	1018
18.6 Einen Stream erzeugen	1019
18.6.1 Stream.of*(...)	1021
18.6.2 Stream.generate(...) und Stream.iterate(...)	1021
18.6.3 Parallele oder sequenzielle Streams	1021
18.7 Terminale Operationen	1022
18.7.1 Die Anzahl der Elemente	1022
18.7.2 Und jetzt alle – forEach*(...)	1022
18.7.3 Einzelne Elemente aus dem Strom holen	1023
18.7.4 Existenztests mit Prädikaten	1024
18.7.5 Einen Strom auf sein kleinstes oder größtes Element reduzieren	1024
18.7.6 Einen Strom mit eigenen Funktionen reduzieren	1025
18.7.7 Stream-Elemente in eine Liste, ein Array oder einen Iterator übertragen	1026
18.7.8 Kollektoren	1027
18.8 Intermediäre Operationen	1029
18.8.1 Element-Vorschau	1029
18.8.2 Filtern von Elementen	1030
18.8.3 Statusbehaftete intermediäre Operationen	1030
18.8.4 Präfix-Operation	1032
18.8.5 Abbildungen	1033
18.8.6 Gatherer	1035
18.9 Zum Weiterlesen	1036

19 Einführung in grafische Oberflächen 1039

19.1 GUI-Frameworks	1039
19.1.1 Kommandozeile	1039
19.1.2 Grafische Benutzeroberfläche	1039
19.2 Abstract Window Toolkit (AWT)	1040
19.2.1 AWT-Frame	1041
19.2.2 Die paint(Graphics)-Methode für den AWT-Frame	1041
19.3 Java Foundation Classes und Swing	1043
19.3.1 Beispielanwendung in Swing	1043
19.4 JavaFX	1045
19.4.1 Fähigkeiten von JavaFX	1045
19.4.2 Die Geschichte von JavaFX: JavaFX 1, JavaFX 2, JavaFX 8, OpenJFX	1046
19.5 Standard Widget Toolkit (SWT) *	1047
19.5.1 Ursprung und Architektur	1047
19.5.2 Einsatz und Erweiterungen	1047
19.5.3 Bewertung und heutige Relevanz	1047
19.6 Zum Weiterlesen	1048

20 Einführung in Dateien und Datenströme 1049

20.1 Alte und neue Welt in java.io und java.nio	1049
20.1.1 java.io-Paket mit File-Klasse	1049
20.1.2 NIO.2 und das java.nio-Paket	1050
20.1.3 java.io.File oder java.nio.*-Typen?	1050
20.2 Dateisysteme und Pfade	1050
20.2.1 FileSystem und Path	1051
20.2.2 Die Utility-Klasse Files	1057
20.3 Dateien mit wahlfreiem Zugriff	1060
20.3.1 Ein RandomAccessFile zum Lesen und Schreiben öffnen	1060
20.3.2 Aus dem RandomAccessFile lesen	1061
20.3.3 Schreiben mit RandomAccessFile	1063
20.3.4 Die Länge des RandomAccessFile	1064
20.3.5 Hin und her in der Datei	1064
20.4 Basisklassen für die Ein-/Ausgabe	1065
20.4.1 Die vier abstrakten Basisklassen	1066

20.4.2	Die abstrakte Basisklasse OutputStream	1066
20.4.3	Die abstrakte Basisklasse InputStream	1069
20.4.4	Die abstrakte Basisklasse Writer	1071
20.4.5	Die Schnittstelle Appendable *	1072
20.4.6	Die abstrakte Basisklasse Reader	1072
20.4.7	Die Schnittstellen Closeable, AutoCloseable und Flushable	1076
20.5	Lesen aus Dateien und Schreiben in Dateien	1077
20.5.1	Byteorientierte Datenströme über Files beziehen	1078
20.5.2	Zeichenorientierte Datenströme über Files beziehen	1079
20.5.3	Die Funktion von OpenOption bei den Files.new*(...)-Methoden	1081
20.5.4	Ressourcen aus dem Klassenpfad und aus JAR-Dateien laden	1083
20.6	Zum Weiterlesen	1085

21 Einführung ins Datenbankmanagement mit JDBC 1087

21.1	Relationale Datenbanken und Java-Zugriffe	1087
21.1.1	Das relationale Modell	1087
21.1.2	Java-APIs zum Zugriff auf relationale Datenbanken	1088
21.1.3	Die JDBC-API und Implementierungen: JDBC-Treiber	1089
21.1.4	H2 ist das Werkzeug auf der Insel	1089
21.2	Eine Beispielabfrage	1090
21.2.1	Schritte zur Datenbankabfrage	1090
21.2.2	Mit Java auf die relationale Datenbank zugreifen	1090
21.3	Zum Weiterlesen	1091

22 Bits und Bytes, Mathematisches und Geld 1093

22.1	Bits und Bytes	1093
22.1.1	Die Bit-Operatoren Komplement, Und, Oder und XOR	1094
22.1.2	Repräsentation ganzer Zahlen in Java – das Zweierkomplement	1095
22.1.3	Das binäre (Basis 2), oktale (Basis 8), hexadezimale (Basis 16) Stellenwertsystem	1097
22.1.4	Auswirkung der Typumwandlung auf die Bit-Muster	1098
22.1.5	Vorzeichenlos arbeiten	1101
22.1.6	Die Verschiebeoperatoren	1103

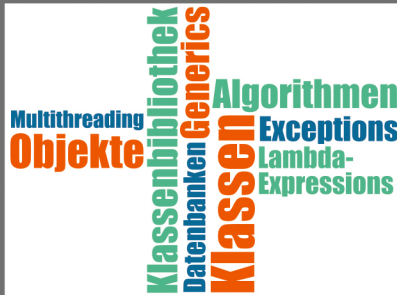
22.1.7	Ein Bit setzen, löschen, umdrehen und testen	1106
22.1.8	Bit-Methoden der Integer- und Long-Klasse	1106
22.2	Gleitkomma-Arithmetik in Java	1108
22.2.1	Spezialwerte für Unendlich, Null, NaN	1109
22.2.2	Standardnotation und wissenschaftliche Notation bei Gleitkommazahlen *	1112
22.2.3	Mantisse und Exponent *	1112
22.3	Die Eigenschaften der Klasse Math	1114
22.3.1	Klassenvariablen der Klasse Math	1115
22.3.2	Absolutwerte und Vorzeichen	1115
22.3.3	Maximum/Minimum	1116
22.3.4	Runden von Werten	1117
22.3.5	Rest der ganzzahligen Division *	1119
22.3.6	Division mit Rundung in Richtung negativ unendlich, alternativer Restwert *	1120
22.3.7	Multiply-Accumulate	1122
22.3.8	Wurzel- und Exponentialmethoden	1122
22.3.9	Der Logarithmus *	1123
22.3.10	Winkelmethode *	1124
22.3.11	Zufallszahlen	1125
22.4	Genauigkeit, Wertebereich eines Typs und Überlaufkontrolle *	1126
22.4.1	Der größte und der kleinste Wert	1126
22.4.2	Überlauf und alles ganz exakt	1127
22.4.3	Was bitte macht eine ulp?	1130
22.5	Zufallszahlen: Random, ThreadLocalRandom und SecureRandom	1131
22.5.1	Die Klasse Random	1131
22.5.2	Die Klasse ThreadLocalRandom	1135
22.5.3	Die Klasse SecureRandom *	1135
22.6	Große Zahlen *	1135
22.6.1	Die Klasse BigInteger	1136
22.6.2	Beispiel: ganz lange Fakultäten mit BigInteger	1143
22.6.3	Große Dezimalzahlen mit BigDecimal	1144
22.6.4	Mit MathContext komfortabel die Rechengenauigkeit setzen	1147
22.6.5	Noch schneller rechnen durch mutable Implementierungen	1148
22.7	Geld und Währung	1149
22.7.1	Geldbeträge repräsentieren	1149
22.7.2	ISO 4217	1149
22.7.3	Währungen in Java repräsentieren	1150
22.8	Zum Weiterlesen	1151

23 Testen mit JUnit	1153
23.1 Softwaretests	1153
23.1.1 Vorgehen beim Schreiben von Testfällen	1154
23.2 Das Test-Framework JUnit	1154
23.2.1 JUnit-Versionen	1155
23.2.2 JUnit aufnehmen	1155
23.2.3 Test-Driven Development und Test-First	1155
23.2.4 Testen, implementieren, testen, implementieren, testen, freuen	1157
23.2.5 JUnit-Tests ausführen	1159
23.2.6 assert*(...)-Methoden der Klasse Assertions	1159
23.2.7 Exceptions testen	1162
23.2.8 Grenzen für Ausführungszeiten festlegen	1163
23.2.9 Beschriftungen mit @DisplayName	1164
23.2.10 Verschachtelte Tests	1164
23.2.11 Tests ignorieren	1164
23.2.12 Mit Methoden der Assumptions-Klasse Tests abbrechen	1165
23.2.13 Parametrisierte Tests	1165
23.3 Java-Assertions-Bibliotheken und AssertJ	1167
23.3.1 AssertJ	1167
23.4 Aufbau größerer Testfälle	1169
23.4.1 Fixtures	1169
23.4.2 Sammlungen von Testklassen und Klassenorganisation	1171
23.5 Wie gutes Design das Testen ermöglicht	1171
23.6 Dummy, Fake, Stub und Mock	1173
23.6.1 Mockito	1174
23.7 JUnit-Erweiterungen, Testzusätze	1176
23.7.1 Webtests	1176
23.7.2 Tests der Datenbankschnittstelle	1176
23.8 Zum Weiterlesen	1177

24 Die Werkzeuge des JDK	1179
24.1 Übersicht	1179
24.1.1 Aufbau und gemeinsame Optionen	1180
24.2 Java-Quellen übersetzen	1180
24.2.1 Der Java-Compiler des JDK	1180
24.2.2 Native Compiler	1181
24.3 Die Java-Laufzeitumgebung	1182
24.3.1 Optionen der JVM	1182
24.3.2 Der Unterschied zwischen java.exe und javaw.exe	1184
24.4 Dokumentationskommentare mit Javadoc	1184
24.4.1 Einen Dokumentationskommentar setzen	1185
24.4.2 Markdown-Syntax und HTML-Tags in Dokumentationskommentaren *	1187
24.4.3 Mit dem Werkzeug javadoc eine Dokumentation erstellen	1188
24.4.4 Generierte Dateien	1188
24.4.5 Dokumentationskommentare im Überblick *	1189
24.4.6 Veraltete (deprecated) Typen und Eigenschaften	1190
24.4.7 Javadoc-Überprüfung mit DocLint	1193
24.4.8 Javadoc und Doclets *	1194
24.5 Das Archivformat JAR	1194
24.5.1 Das Dienstprogramm jar benutzen	1195
24.5.2 Das Manifest	1195
24.5.3 Applikationen in JAR-Archiven starten; ausführbare JAR-Dateien	1196
24.6 Zum Weiterlesen	1197
 Anhang	 1199
A Java SE-Module und Paketübersicht	1199
B Referenz Schlüsselwörter	1217
 Index	 1223

Java-Programmierung von A bis Z

Im Java-Kultbuch lehrt Christian Ullenboom anschaulich und praxisorientiert alles Wissenswerte zu Klassen, Objekten, Generics und Lambda-Ausdrücken. Kompakte Einführungen in Spezialthemen runden das Buch ab. Die Insel ist besonders geeignet für Leserinnen und Leser mit Grundkenntnissen in der Programmierung, Studierende und Umsteiger von anderen Sprachen.



Umfassendes Java-Wissen

Warum übersetzt der Compiler Folgendes ohne Murren?

Listing 2.36 src/main/java/WithoutComplain.java

```
void main() {  
    http://www.tutego.de/  
    IO.print( "IT training institute" );  
}
```

Viele Beispiele und Tipps



Mit Spaß lernen und verstehen

Java-Grundlagen verstehen

Hier lernen Sie die Sprache Java mit all ihren Merkmalen und Möglichkeiten kennen. Gründlich, detailliert und mit einer Prise Humor erklärt. Mit hilfreichen Informationen zur Entwicklungsumgebung IntelliJ IDEA.

Praxiserprobte Programme und viele Tipps

Profitieren Sie von praxisnahen Beispielprogrammen und zahlreichen Expertentipps für den Programmieralltag. Die Insel ist stets up to date. Diese Auflage ist aktuell zur LTS-Version Java 25.

Lehrbuch, Nachschlagewerk und Referenz in einem

Dieses Standardwerk bietet Einsteigern wie Fortgeschrittenen eine systematische Einführung mit vielen Hintergrundinformationen. Die klar strukturierte API-Dokumentation macht die Insel zur unverzichtbaren Referenz.



Alle Beispielprogramme stehen zum Download bereit.



Christian Ullenboom, Diplom-Informatiker, ist erfahrener Java-Trainer und Gründer des IT-Schulungsunternehmens tutego. 2005 wurde er in Oracles Java Champions Board aufgenommen. Wenn er nicht auf Java unterwegs ist, bereist er andere Teile der Welt. Besuchen Sie ihn auf seiner Website ullenboom.de.

Aus dem Inhalt

Java-Grundlagen

- Imperative Sprachkonzepte
- Klassen und Objekte
- Eigene Klassen schreiben
- Ausnahmebehandlung
- Besondere Typen der Java SE
- Generics<T>
- Lambda-Ausdrücke und funktionale Programmierung
- Architektur und Design
- Records und Sealed Classes
- Klassenbibliothek

Spezialthemen

- Nebenläufige Programmierung
- Datenstrukturen und Algorithmen
- Grafische Oberflächen
- Dateien und Datenströme
- Datenbankmanagement mit JDBC
- Testen mit JUnit

