

Christian Ullenboom



Java

ist auch eine Insel

Einführung, Ausbildung, Praxis

Das
Standard-
werk

- ▶ Programmieren mit der Java Platform, Standard Edition 21
- ▶ Java von A bis Z: Einführung, Praxis, Referenz
- ▶ Von Ausdrücken und Anweisungen zu Klassen und Objekten

17., 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 (der 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 Grundssäulen 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.

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 gibt es nicht erst seit den objektorientierten Programmiersprachen, objektorientierte Programmiersprachen erleichtern aber die Programmierung wiederverwendbarer Softwarekomponenten. So sind auch die vielen Tausend Klassen der Bibliothek ein Beispiel dafür, dass sich Entwickler nicht ständig um die Umsetzung etwa von Datenstrukturen oder um die Pufferung von Datenströmen kümmern 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 auch in einer einfachen Programmiersprache wie C lässt sich 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; einige Autoren nennen allerdings nur Attribute Eigenschaften. Welche Eigenschaften eine Klasse tatsächlich besit-

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 *instanzieren* tritt das einfache Wort *erzeugen*. *Instanz* ist eine irreführende Übersetzung des englischen Ausdrucks *»instance«*.

zen soll, wird in der Analyse- und Designphase festgesetzt. 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 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/21/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.

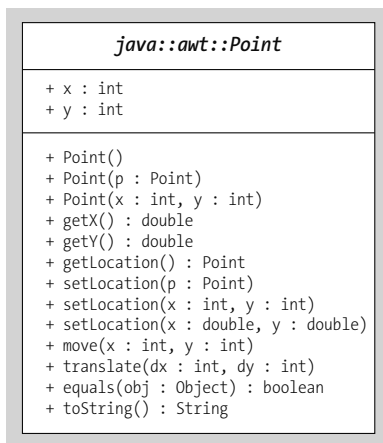


Abbildung 3.1 Die Klasse »`java.awt.Point`« in der UML-Darstellung

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 ihr Einsatzgebiet 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 Personen-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 Exemplare erzeugen lassen und wie der Zugriff auf die Eigenschaften der `Point`-Objekte aussieht.

3.4.1 Ein Exemplar 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 Exemplar 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 Entwickler diese Schreibweise auch abkürzen können.) 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.

Könnte die Speicherverwaltung von Java für das anzulegende Objekt freien Speicher reservieren und konnte der Konstruktor gütig 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.



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 uninitialized):

```
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 Punkt-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.



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.

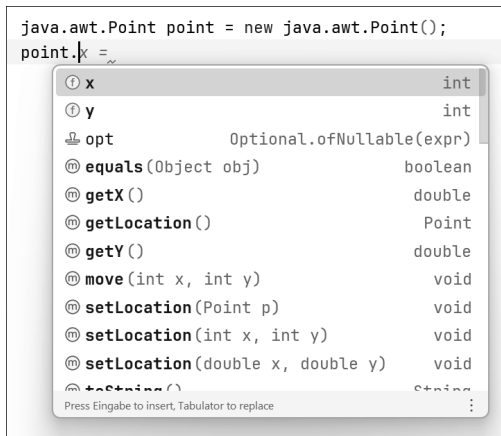


Abbildung 3.3 Die Tastenkombination `[Strg] + Leertaste` zeigt an, welche Eigenschaften eine Referenz ermöglicht. Eine Auswahl mit der `[↵]`-Taste wählt die Eigenschaft aus und setzt insbesondere bei Methoden den Cursor zwischen das Klammerpaar.

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 ja `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:



```
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 `PlayerAndDoorAsPoints.java`

```
class PlayerAndDoorAsPoints {  
  
    public static void main( String[] args ) {  
        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 );  
  
        System.out.println( player.distance( door ) ); // 90.0  
    }  
}
```

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.



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.

toString()

Die Methode `toString()` liefert als Ergebnis ein `String`-Objekt, das den Zustand des Punktes 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 PointToStringDemo.java

```

class PointToStringDemo {

    public static void main( String[] args ) {
        java.awt.Point player = new java.awt.Point();
        java.awt.Point door   = new java.awt.Point();
        door.setLocation( 10, 100 );

        System.out.println( player.toString() ); // java.awt.Point[x=0,y=0]
        System.out.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();
System.out.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:

```
System.out.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();
System.out.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();
System.out.println( p.toString().length() ); // 23
```

genau das Gleiche bewirkt wie:

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

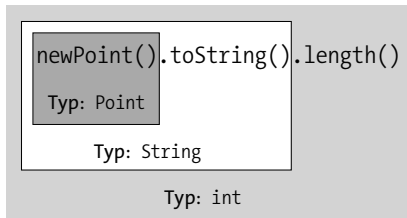


Abbildung 3.5 Jede Schachtelung ergibt einen neuen Typ.

Im Prinzip funktioniert auch Folgendes:

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

Dies ist hier allerdings unsinnig, da zwar das Objekt erzeugt und eine Objektvariable gesetzt wird, anschließend das Objekt aber für die automatische Speicherbereinigung wieder freigegeben 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 Der Zusammenhang von new, Heap und Garbage-Collector

Bekommt das Laufzeitsystem die Anfrage, ein Objekt mit `new` zu erzeugen, so reserviert es so viel Speicher, dass alle Objekteigenschaften und Verwaltungsinformationen dort Platz finden. Ein `Point`-Objekt speichert die Koordinaten in zwei `int`-Werten, also sind mindestens 2 mal 4 Byte nötig. Den Speicherplatz nimmt die Laufzeitumgebung vom *Heap*. Der Heap wächst von einer Startgröße bis hin zu einer erlaubten Maximalgröße, damit ein Java-Programm nicht beliebig viel Speicher vom Betriebssystem abgreifen kann, was die Maschine möglicherweise in den Ruin treibt. In der HotSpot JVM ist der Heap zum Start $\frac{1}{64}$ des Hauptspeichers groß und wächst dann bis zur maximalen Größe von $\frac{1}{4}$ des Hauptspeichers.⁶

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 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.

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.⁷

Heap und Stack

Die JVM-Spezifikation sieht für Daten fünf verschiedene Speicherbereiche (engl. *runtime data areas*) vor.⁸ Neben dem *Heap-Speicher* wollen wir uns den *Stack-Speicher* (Stapelspeicher) kurz anschauen. Den nutzt die Java-Laufzeitumgebung zum Beispiel für lokale Variablen. Auch verwendet Java den Stack beim Methodenaufruf mit Parametern. Die Argumente kommen vor dem Methodenaufruf auf den Stapel, und die aufgerufene Methode kann über den Stack auf die Werte lesend oder schreibend zugreifen. Bei endlosen rekursiven Methodenaufrufen ist irgendwann die maximale Stack-Größe erreicht, und es kommt zu einer `Exception` vom Typ `java.lang.StackOverflowError`. Da mit jedem Thread ein JVM-Stack assoziiert ist, bedeutet das das Ende des Threads, wobei andere Threads unbeeindruckt weiterlaufen.

⁶ <https://docs.oracle.com/en/java/javase/21/gctuning/ergonomics.html>

⁷ 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.

⁸ § 2.5 der JVM-Spezifikation, <https://docs.oracle.com/javase/specs/jvms/se21/html/jvms-2.html#jvms-2.5>



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

Nehmen wir folgendes Szenario an:

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

Wir deklarieren eine `Point`-Variable, bauen ein Exemplar auf und belegen die Variable. 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.⁹ 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 wird, ist es auch schon tot. Es gibt verschiedene GC-Algorithmen, und jeder Hersteller einer JVM hat eigene Verfahren.

3.4.5 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`

- `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.

⁹ Mit dem gesetzten `java`-Schalter `-verbose:gc` gibt es immer Konsolenausgaben, wenn der GC nicht mehr referenzierte Objekte erkennt und wegräumt.

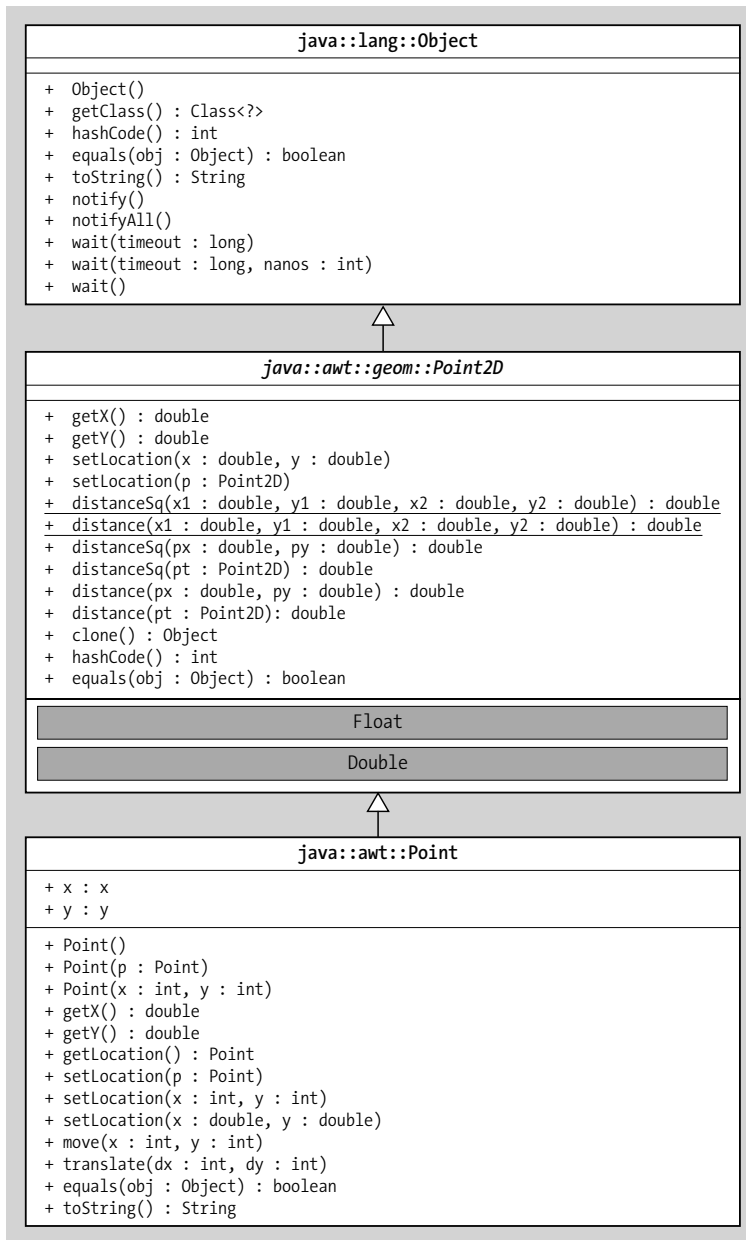


Abbildung 3.6 Vererbungshierarchie bei Point2D

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 ein sehr ausführliches 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. Für Entwickler ist es unabdingbar, 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:¹⁰

```
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 Punktes zu angegebenen Koordinaten.
- `double distance(Point2D pt)`
Berechnet den Abstand des aktuellen Punktes zu den Koordinaten des übergebenen Punktes.

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:

¹⁰ Damit ist die Klassendeklaration noch nicht vollständig, da ein `implements Serializable` fehlt, doch das soll uns jetzt erst einmal egal sein.

Listing 3.3 PointEqualsDemo.java

```

class PointEqualsDemo {

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

        System.out.println( player.equals( door ) );    // true
        System.out.println( door.equals( player ) );    // true

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

        System.out.println( snake.equals( door ) );    // false
    }
}

```

Da Spieler und Tür die gleichen Koordinaten 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.

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 Distances.java

```

class Distances {

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

        System.out.println( player.distance( door ) );           // 0.0
        System.out.println( player.distance( snake ) );         // 10.0
        System.out.println( player.distance( snake.x, snake.y ) ); // 10.0
    }
}

```

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

3.4.6 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.



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 ZZZZZnake.java

```
public class ZZZZZnake {

    public static void main( String[] args ) {
        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 ) )
            System.out.print( '&' );
        else if ( snakePosition.equals( p ) )
            System.out.print( 'S' );
        else if ( goldPosition.equals( p ) )
            System.out.print( '$' );
        else if ( doorPosition.equals( p ) )
            System.out.print( '#' );
        else System.out.print( '.' );
    }
    System.out.println();
}

// Determine status

if ( rich && playerPosition.equals( doorPosition ) ) {
    System.out.println( "You won!" );
    return;
}
if ( playerPosition.equals( snakePosition ) ) {
    System.out.println( "SSSSSS. 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.

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 5 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 Klassenbibliothek von Java ist mit Tausenden Typen sehr umfangreich und deckt alles ab, was Entwickler von plattformunabhängigen Programmen als Basis benötigen. Dazu gehören Datenstrukturen, Klassen zur Datums-/Zeitberechnung, Dateiverarbeitung usw. Die meisten Typen sind in Java selbst implementiert (und der Quellcode ist in der Regel aus der Entwicklungsumgebung direkt verfügbar), aber einige Teile sind nativ implementiert, etwa wenn es darum geht, aus einer Datei zu lesen.

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 Fließkommazahlen 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 Nutzer immer der Fall –, muss sie dem Compiler mit der gesamten Paketangabe bekannt gemacht werden. Hierzu reicht der Klassenname allein nicht aus, denn es kann ja 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> class AwtWithoutImport { public static void main(String[] args){ 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); System.out.println(p); System.out.println(t.contains(15, ↵ 15)); } } </pre>	Listing 3.7 AwtWithImport.java <pre> import java.awt.Point; import java.awt.Polygon; class AwtWithImport { public static void main(String[] args){ Point p = new Point(); Polygon t = new Polygon(); t.addPoint(10, 10); t.addPoint(10, 20); t.addPoint(20, 10); System.out.println(p); System.out.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 unabhk6mmliche Qualifizierung gegeben.



Hinweis

Die Typen aus `java.lang` sind automatisch importiert, sodass z. B. ein `import java.lang.String;` nicht n6tig ist.

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

Greift eine Java-Klasse auf mehrere andere Typen des gleichen Pakets zur6ck, kann die Anzahl der `import`-Deklarationen gro6 werden. In unserem Beispiel nutzen wir mit `Point` und `Polygon` nur zwei Klassen aus `java.awt`, aber es l6sst sich schnell ausmalen, was passiert, wenn aus dem Paket f6r grafische Oberfl6chen zus6tzlich Fenster, Beschriftungen, Schaltfl6chen, 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 Typen im Paket `java.awt` und `java.math`, sodass der Compiler das Paket f6r die Klassen `Point` und `Polygon` zuordnen kann, wie auch das Paket f6r die Klasse `BigInteger`.



Hinweis

Das `*` ist nur auf der letzten Hierarchieebene erlaubt und gilt immer f6r 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 schlie6t die Unterverzeichnisse nicht ein.

Das `*` verk6rzt 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 6bersetzungsfehler, weil der Compiler nicht wei6t, was gemeint ist. Eine volle Qualifizierung l6st 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 Hierarchische Strukturen über Pakete und die Spiegelung im Dateisystem

Die zu einem Paket gehörenden Klassen befinden sich normalerweise¹¹ im gleichen Verzeichnis. Der Name des Pakets ist gleich dem Namen des Verzeichnisses (und natürlich umgekehrt). Statt des Verzeichnistrenners (etwa `»/«` oder `»\«`) steht ein Punkt.

Nehmen wir folgende Verzeichnisstruktur mit einer Hilfsklasse an:

com/tutego/insel/printer/DatePrinter.class

Hier ist der Paketname `com.tutego.insel.printer` und somit der Verzeichnisname *com/tutego/insel/printer*. Umlaute und Sonderzeichen sollten vermieden werden, da sie auf dem Dateisystem immer wieder für Ärger sorgen. Aber Bezeichner sollten ja sowieso immer auf Englisch sein.

Der Aufbau von Paketnamen

Prinzipiell kann ein Paketname beliebig sein, doch Hierarchien bestehen in der Regel aus umgedrehten Domännennamen. Aus der Domäne zur Website <http://tutego.com> wird also `com.tutego`. Diese Namensgebung gewährleistet, dass Klassen auch weltweit eindeutig bleiben. Ein Paketname wird in aller Regel komplett kleingeschrieben.

3.6.6 Die package-Deklaration

Um die Klasse `DatePrinter` in ein Paket `com.tutego.insel.printer` zu setzen, müssen zwei Dinge gelten:

- Sie muss sich physisch in einem Verzeichnis befinden, also in *com/tutego/insel/printer*.
- Der Quellcode enthält zuoberst eine `package`-Deklaration.

¹¹ Ich schreibe »normalerweise«, 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.

Die package-Deklaration muss ganz am Anfang stehen, sonst gibt es einen Übersetzungsfehler (selbstverständlich lassen sich Kommentare vor die package-Deklaration setzen):

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

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

import java.time.LocalDate;
import java.time.format.*;

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

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

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

```
public class DatePrinterUser1 {
    public static void main( String[] args ) {
        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;

public class DatePrinterUser2 {
    public static void main( String[] args ) {
        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 und die Anpassungen an den `import`- und `package`-Deklarationen übernimmt eine moderne IDE für uns.

3.6.7 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 auch feinere Sichtbarkeiten, und Konflikte mit anderen Unternehmen und Autoren werden vermieden. 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.

Stände 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
```

3.6.8 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 Quelldatei; die Codezeilen könnten grundsätzlich auch aus einer Datenbank kommen oder zur Laufzeit generiert werden.

3.6.9 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

Binde für die Bildschirmausgabe die statische Variable `out` aus `System` statisch ein:

```
import static java.lang.System.out;
```

Bei der sonst üblichen Ausgabe über `System.out.print*(...)` kann nach dem statischen Import der Klassenname entfallen, und es bleibt beim `out.print*(...)`.

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

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

```
package com.tutego.insel.oo;

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;

class StaticImport {

    public static void main( String[] args ) {
        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 typen-

los, kann also jeder Referenzvariablen zugewiesen und jeder Methode übergeben werden, die ein Objekt erwartet.¹²



Beispiel

Deklaration und Initialisierung zweier Objektvariablen mit `null`:

```
Point p = null;
String s = null;
System.out.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.¹³

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`¹⁴ aus:

¹² `null` verhält sich also so, als ob es ein Untertyp jedes anderen Typs wäre.

¹³ Hier unterscheiden sich C++ und Java.

¹⁴ 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/ooop/NullPointer.java

```

package com.tutego.insel.ooop;           // 1
public class NullPointer {               // 2
    public static void main( String[] args ) { // 3
        java.awt.Point p = null;         // 4
        String          s = null;         // 5
        p.setLocation( 1, 2 );           // 6
        s.length();                      // 7
    }                                     // 8
}                                         // 9

```

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 com.tutego.insel.ooop.NullPointer.main(NullPointer.java:6)

```

Die Laufzeitumgebung teilt uns in der Fehlermeldung mit, dass sich der Fehler, die `NullPointerException`, in Zeile 6 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 Stringlänge ausgeben, wenn ein `String` eingegeben wurde:

Listing 3.13 `src/main/java/NullCheck.java`, `main`

```
String s = javax.swing.JOptionPane.showInputDialog( "Input a string" );
if ( s != null && ! s.isEmpty() )
    System.out.println( "Length of string: " + s.length() );
else
    System.out.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` *

Da `null` viel zu oft vorkommt, `null`-Referenzierungen aber vermieden werden müssen, gibt es viel Code der Art `o != null ? o : non_null_o`. Diverse Programmiersprachen wie JavaScript, Kotlin, Objective-C, PHP oder Swift bieten für dieses Konstrukt eine Abkürzung über den sogenannten *null coalescing operator* (*coalescing* heißt auf Deutsch »verschmelzend«). Er wird mal als `??` oder als `?:` geschrieben, für unser Beispiel so: `o ?? non_null_o`. Besonders hübsch ist das bei sequenziellen 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*.

**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 ItsTheSame.java, main

```
public static void main( String[] args ) {
    Point p = new Point();
    Point q = p;
    p.x = 10;
    System.out.println( q.x ); // 10
}
```



```
q.y = 5;
System.out.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/com/tutego/insel/oop/DrawPlayerAndSnake.java`

```
package com.tutego.insel.oop;
import java.awt.Point;

public class DrawPlayerAndSnake {

    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 )
                    System.out.print( 'S' );
                else if ( playerPosition.distanceSq( x, y ) == 0 )
                    System.out.print( '&' );
                else System.out.print( '.' );
            }
        }
    }
}
```

```

        System.out.println();
    }
}

public static void main( String[] args ) {
    Point playerPosition = new Point();
    Point snakePosition  = new Point();
    System.out.println( playerPosition );
    System.out.println( snakePosition );
    initializePosition( playerPosition );
    initializePosition( snakePosition );
    System.out.println( playerPosition );
    System.out.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 ja 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 `JavalsAlwaysCallByValue.java`

```
package com.tutego.insel.oop;
import java.awt.Point;

public class JavaIsAlwaysCallByValue {

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

    public static void main( String[] args ) {
        Point p = new Point( 10, 20 );
        clear( p );
        System.out.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 ja nur ein temporärer alternativer Name für das `p` aus `main`; eine Neuweisung 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 dem Programmierer vorgeben, dass er Variablen nicht wieder beschreiben darf. 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.

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 );
System.out.println( p == q ); // true, da p und q dasselbe Objekt referenzieren
System.out.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 );
System.out.println( p == q );           // false
System.out.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`, Ausschnitt

```
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 Punktobjekts, das als Argument an `equals(...)` übergeben wurde.

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

Glücklicherweise müssen wir als Programmierer nicht lange darüber nachdenken, 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`, Ausschnitt

```
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 den Identitätsvergleich und Gleichwertigkeitstest eigene Operatoren anbieten. Was bei Java `==` und `equals(...)` sind, sind bei Python `is` und `==`, bei Swift `===` und `==`.

3.8 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 Programmierbibliothek. Eine plattformunabhängige Sprache – so wie sich viele C oder C++ vorstellen – ist nicht wirklich plattformunabhängig, wenn auf jedem Rechner andere Funktionen und Programmiermodelle eingesetzt werden. Genau dies ist der Schwachpunkt von C++. Die Algorithmen, die kaum vom Betriebssystem abhängig sind, lassen sich überall gleich anwenden, doch spätestens bei der Ein-/Ausgabe oder grafischen Oberflächen ist Schluss. Die Java-Bibliothek dagegen versucht, von den plattformspezifischen Eigenschaften zu abstrahieren, und die Entwickler haben sich große Mühe gegeben, alle wichtigen Methoden in wohlgeformten objektorientierten Klassen und Paketen unterzubringen. Diese decken insbesondere die zentralen Bereiche Datenstrukturen, Ein- und Ausgabe, Grafik- und Netzwerkprogrammierung ab.

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 API der *Java Platform, Standard Edition* (»Java SE«) besteht aus folgenden Modulen, die alle mit dem Präfix `java` beginnen.

Modul	Beschreibung
<code>java.base</code>	fundamentale Typen der Java SE-Plattform
<code>java.compiler</code>	Java-Sprachmodell, Annotationsverarbeitung, Java-Compiler-API
<code>java.datatransfer</code>	API für den Datentransfer zwischen Applikationen, in der Regel die Zwischenablage
<code>java.desktop</code>	grafische Oberflächen mit AWT und Swing, <i>Accessibility-API</i> , Audio, Drucken und JavaBeans
<code>java.instrument</code>	Instrumentalisierung ist die Veränderung der Java-Programme zur Laufzeit.
<code>java.logging</code>	<i>Logging-API</i>
<code>java.management</code>	<i>Java Management Extensions (JMX)</i>
<code>java.management.rmi</code>	RMI-Connector für den Remote-Zugriff auf die JMX-Beans
<code>java.naming</code>	<i>Java Naming and Directory Interface (JNDI) API</i>
<code>java.prefs</code>	Die <i>Preferences-API</i> dient zum Speichern von Benutzereinstellungen.
<code>java.rmi</code>	entfernte Methodenaufrufe; <i>Remote Method Invocation (RMI) API</i>
<code>java.scripting</code>	<i>Scripting-API</i>
<code>java.security.jgss</code>	Java-Binding der <i>IETF Generic Security Services API (GSS-API)</i>
<code>java.security.sasl</code>	Java-Unterstützung für <i>IETF Simple Authentication and Security Layer (SASL)</i>
<code>java.sql</code>	JDBC-API für den Zugriff auf relationale Datenbanken
<code>java.sql.rowset</code>	<i>JDBC-RowSet-API</i>
<code>java.xml</code>	XML-Klassen: <i>Java API for XML Processing (JAXP)</i> , <i>Streaming API for XML (StAX)</i> , <i>Simple API for XML (SAX)</i> , <i>W3C Document Object Model (DOM) API</i>
<code>java.xml.crypto</code>	API für XML-Kryptografie

Tabelle 16.1 Module der Java SE

Das `java.base`-Modul ist das wichtigste Modul, und es 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 schön grafisch dar (siehe Abbildung 16.1).

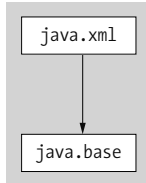


Abbildung 16.1 Das Modul »`java.xml`« hat eine Abhängigkeit zum »`java.base`«-Modul.

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

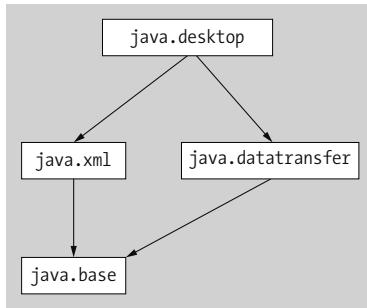


Abbildung 16.2 Abhängigkeiten des Moduls »`java.desktop`«

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 so eine Konstruktion ist *Aggregator-Modul*. Das `java.se`-Modul definiert auf diese Weise die API für die Java SE-Plattform (siehe Abbildung 16.3).

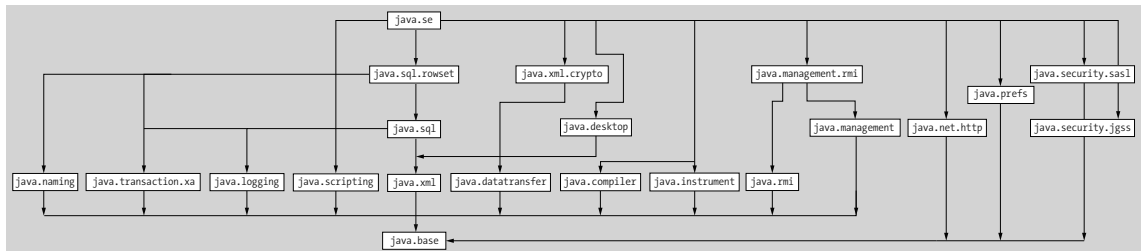


Abbildung 16.3 Abhängigkeiten des Moduls »`java.se`«



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.

Weitere Module

Ein weiteres Modul, das ebenfalls mit `java` beginnt, aber nicht zum Java SE-Standard zählt, ist `java.smartcardio` (Java-API für die Kommunikation mit Smart Cards nach ISO/IEC 7816-4).

Das OpenJDK ist die Standardimplementierung der Java SE. Es liefert den Entwicklern weitere Pakete und Klassen, etwa mit einem HTTP-Server oder den Java-Werkzeugen wie dem Java-Compiler und dem *javadoc*-Tool. Es gibt mehrere Module, die alle mit dem Präfix `jdk` beginnen.

16.1.2 Übersicht über die Pakete der Standardbibliothek

Die *Java 11 Core Java SE API* besteht aus folgenden Modulen und Paketen – eine Kurzbeschreibung findet sich im Anhang:

Module	enthaltene Pakete
<code>java.base</code>	<code>java.io</code> , <code>java.lang</code> , <code>java.lang.annotation</code> , <code>java.lang.invoke</code> , <code>java.lang.module</code> , <code>java.lang.ref</code> , <code>java.lang.reflect</code> , <code>java.math</code> , <code>java.net</code> , <code>java.net.spi</code> , <code>java.nio</code> , <code>java.nio.channels</code> , <code>java.nio.channels.spi</code> , <code>java.nio.charset</code> , <code>java.nio.charset.spi</code> , <code>java.nio.file</code> , <code>java.nio.file.attribute</code> , <code>java.nio.file.spi</code> , <code>java.security</code> , <code>java.security.cert</code> , <code>java.security.interfaces</code> , <code>java.security.spec</code> , <code>java.text</code> , <code>java.text.spi</code> , <code>java.time</code> , <code>java.time.chrono</code> , <code>java.time.format</code> , <code>java.time.temporal</code> , <code>java.time.zone</code> , <code>java.util</code> , <code>java.util.concurrent</code> ,
<code>java.base</code>	<code>java.util.concurrent.atomic</code> , <code>java.util.concurrent.locks</code> , <code>java.util.function</code> , <code>java.util.jar</code> , <code>java.util.regex</code> , <code>java.util.spi</code> , <code>java.util.stream</code> , <code>java.util.zip</code> , <code>javax.crypto</code> , <code>javax.crypto.interfaces</code> , <code>javax.crypto.spec</code> , <code>javax.net</code> , <code>javax.net.ssl</code> , <code>javax.security.auth</code> , <code>javax.security.auth.callback</code> , <code>javax.security.auth.login</code> , <code>javax.security.auth.spi</code> , <code>javax.security.auth.x500</code> , <code>javax.security.cert</code>

Tabelle 16.2 Übersicht über die Pakete in den Modulen der Java 21 Core Java SE API

Module	enthaltene Pakete
java.compiler	javax.annotation.processing, javax.lang.model, javax.lang.model.element, javax.lang.model.type, javax.lang.model.util, javax.tools
java.datatransfer	java.awt.datatransfer
java.desktop	java.applet, java.awt, java.awt.color, java.awt.desktop, java.awt.dnd, java.awt.event, java.awt.font, java.awt.geom, java.awt.im, java.awt.im.spi, java.awt.image, java.awt.image.renderable, java.awt.print, java.beans, java.beans.beancontext, javax.accessibility, javax.imageio, javax.imageio.event, javax.imageio.metadata, javax.imageio.plugins.bmp, javax.imageio.plugins.jpeg, javax.imageio.plugins.tiff, javax.imageio.spi, javax.imageio.stream, javax.print, javax.print.attribute, javax.print.attribute.standard, javax.print.event, javax.sound.midi, javax.sound.midi.spi, javax.sound.sampled, javax.sound.sampled.spi, javax.swing, javax.swing.border, javax.swing.colorchooser, javax.swing.event, javax.swing.filechooser, javax.swing.plaf, javax.swing.plaf.basic, javax.swing.plaf.metal, javax.swing.plaf.multi, javax.swing.plaf.nimbus, javax.swing.plaf.synth, javax.swing.table, javax.swing.text, javax.swing.text.html, javax.swing.text.html.parser, javax.swing.text.rtf, javax.swing.tree, javax.swing.undo
java.instrument	java.lang.instrument
java.logging	java.util.logging
java.management	java.lang.management, javax.management, javax.management.loading, javax.management.modelmbean, javax.management.monitor, javax.management.openmbean, javax.management.relation, javax.management.remote, javax.management.timer
java.management.rmi	javax.management.remote.rmi
java.naming	javax.naming, javax.naming.directory, javax.naming.event, javax.naming.ldap, javax.naming.spi
java.prefs	java.util.prefs
java.rmi	java.rmi, java.rmi.activation, java.rmi.dgc, java.rmi.registry, java.rmi.server, javax.rmi.ssl

Tabelle 16.2 Übersicht über die Pakete in den Modulen der Java 21 Core Java SE API (Forts.)

Module	enthaltene Pakete
java.scripting	javax.script
java.security.jgss	javax.security.auth.kerberos, org.ietf.jgss
java.security.sasl	javax.security.sasl
java.sql	java.sql, javax.sql, javax.transaction.xa
java.sql.rowset	javax.sql.rowset, javax.sql.rowset.serial, javax.sql.rowset.spi
java.xml	javax.xml, javax.xml.catalog, javax.xml.datatype, javax.xml.namespace, javax.xml.parsers, javax.xml.stream, javax.xml.stream.events, javax.xml.stream.util, javax.xml.transform, javax.xml.transform.dom, javax.xml.transform.sax, javax.xml.transform.stax, javax.xml.transform.stream, javax.xml.validation, javax.xml.xpath, org.w3c.dom, org.w3c.dom.bootstrap, org.w3c.dom.events, org.w3c.dom.ls, org.w3c.dom.ranges, org.w3c.dom.views, org.xml.sax, org.xml.sax.ext, org.xml.sax.helpers
java.xml.crypto	javax.xml.crypto, javax.xml.crypto.dom, javax.xml.crypto.dsig, javax.xml.crypto.dsig.dom, javax.xml.crypto.dsig.keyinfo, javax.xml.crypto.dsig.spec

Tabelle 16.2 Übersicht über die Pakete in den Modulen der Java 21 Core Java SE API (Forts.)

Entwickler sollten folgende Pakete von deren Möglichkeiten her zuordnen können:

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.

Tabelle 16.3 Wichtige Pakete in der Java SE

Paket	Beschreibung
<code>java.lang</code>	Ein Paket, das automatisch eingebunden ist. Enthält unverzichtbare Klassen wie <code>String</code> -, <code>Thread</code> - oder <code>Wrapper</code> -Klassen.
<code>java.net</code>	Kommunikation über Netzwerke. Bietet Klassen zum Aufbau von Client- und Serversystemen, die sich über TCP bzw. IP mit dem Internet verbinden lassen.
<code>java.text</code>	Unterstützung für internationalisierte Programme. Bietet Klassen zur Behandlung von Text und zur Formatierung von Datumswerten und Zahlen.
<code>java.util</code>	Bietet Typen für Datenstrukturen, Raum und Zeit sowie für Teile der Internationalisierung sowie für Zufallszahlen. Unterpakete kümmern sich um reguläre Ausdrücke und Nebenläufigkeit.
<code>javax.swing</code>	Swing-Komponenten für grafische Oberflächen. Das Paket besitzt diverse Unterpakete.

Tabelle 16.3 Wichtige Pakete in der Java SE (Forts.)

Eine vollständige Übersicht aller Pakete gibt der Anhang, »Java SE-Module und Paketübersicht«. Als Entwickler ist es unumgänglich, für die Details die Java-API-Dokumentation unter <https://docs.oracle.com/en/java/javase/21/docs/api/index.html> zu studieren.

Offizielle Schnittstelle (java- und javax-Pakete)

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

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

Im Endeffekt haben Entwickler es mit folgenden Bibliotheken zu tun:

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

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

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 System-Zeitgeber. Sie hat keinen Bezugspunkt zu irgendeinem Datum.

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

Tipp

Die Werte von `nanoTime()` sind immer aufsteigend, was für `currentTimeMillis()` nicht zwingend gelten muss, da sich Java die Zeit vom Betriebssystem holt, und da kann sich die Systemzeit ändern, etwa wenn der Benutzer die Zeit anpasst. Differenzen von `currentTimeMillis()`-Zeitstempeln sind dann komplett falsch und könnten sogar negativ sein.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 Programm des JDK und integriert einen freien Profiler. *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 Klassen-Browser schreiben. Dieser soll alle zum laufenden Programm gehörenden Klassen und darüber hinaus weitere Informationen anzeigen, wie etwa Variablenbelegung, deklarierte Methoden, Konstruktoren und Informationen über die Vererbungshierarchie. Dafür benötigen wir die Bibliotheksklasse `Class`. Exemplare der Klasse `Class` sind Objekte, die zum Beispiel eine Java-Klasse, ein Record oder 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.

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

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.

16.3.1 An ein Class-Objekt kommen

Zunächst müssen wir für eine bestimmte Klasse das zugehörige Class-Objekt in Erfahrung bringen. Class-Objekte selbst kann nur die JVM erzeugen. Wir können das nicht (die Objekte sind immutable, 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.
- ▶ 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 Klassenmethode `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.util.Date` heranzukommen:

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

```
Class<Date> c1 = java.util.Date.class;
System.out.println( c1 );           // class java.util.Date
Class<?> c2 = new java.util.Date().getClass();
// oder Class<? extends Date> c2 = ...

System.out.println( c2 );           // class java.util.Date
try {
    Class<?> c3 = Class.forName( "java.util.Date" );
    System.out.println( c3 );       // class java.util.Date
}
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 eingängiger, und der Compiler kann prüfen, ob es den Typ gibt. Eine volle Qualifizierung ist nötig, `Class.forName("Date")` würde nur nach `Date` in dem Default-Paket suchen – die Rückgabe ist ja keine Sammlung.



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,
TypeDescriptor.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 noch 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.

Während `ClassNotFoundException` häufiger vorkommt als `NoClassDefFoundError`, ist die Ausnahme im Allgemeinen ein Indiz dafür, dass ein Java-Archiv im Modulpfad 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 und 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 ein 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 etwa 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.

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«

Bemerkung

Insgesamt machen die Klassen `System` und `Runtime` keinen besonders aufgeräumten Eindruck (siehe Abbildung 16.4); sie wirken irgendwie 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 Frage nach dem Speicherstand oder der Anzahl der Prozessoren beantwortet `Runtime` mit `availableProcessors()`, aber auch `MBeans`, wie etwa `ManagementFactory.getOperatingSystemMXBean().getAvailableProcessors()`. Aber API-Design ist wie Sex: Eine unüberlegte Aktion, und die Brut lebt mit uns für immer.

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 das, was aktuell genutzt wird und bis auf `maxMemory()` wachsen kann. Es kann prinzipiell auch wieder schrumpfen. Es gilt: `maxMemory() > totalMemory()`.
- ▶ `freeMemory()` ist das, was frei für neue Objekte ist und die automatische Speicherbereinigung auch wieder anhebt. Es gilt: `totalMemory() > freeMemory()`. Allerdings ist `freeMemory()` nicht der gesamte freie verfügbare Speicherbereich, denn es fehlt noch der »Anteil« von `maxMemory()`.

Zwei Informationen fehlen also, die berechnet werden müssen:

Benutzter Speicher:

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

Freier Gesamtspeicher:

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



Beispiel

Gib Informationen über den Speicher auf einem Rechner aus:

```
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=126 MiB, free=124 MiB, max=2016 MiB, used=1 MiB, total free=2014 MiB
```

16.4.2 Anzahl der CPUs bzw. Kerne

Die Runtime-Methode `availableProcessors()` liefert die Anzahl logischer Prozessoren bzw. Kerne.



Beispiel

```
System.out.println( Runtime.getRuntime().availableProcessors() ); // 4
```


16.4.3 Systemeigenschaften der Java-Umgebung

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

Beispiel

Gib den Namen des Betriebssystems aus:

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

Gib alle Systemeigenschaften auf dem Bildschirm aus:

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

Die Ausgabe beginnt mit:

```
-- listing properties --
sun.desktop=windows
awt.toolkit=sun.awt.windows.WToolkit
java.specification.version=9
file.encoding.pkg=sun.io
sun.cpu.isalist=amd64
...
```



Tabelle 16.4 zeigt eine Liste der wichtigsten Standardsystemeigenschaften:

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

Tabelle 16.4 Standardsystemeigenschaften

Schlüssel	Bedeutung
user.home	Home-Verzeichnis des Benutzers
user.dir	aktuelles Verzeichnis des Benutzers

Tabelle 16.4 Standardsystemeigenschaften (Forts.)

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.

16.4.4 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(String[])`-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 16.2 `com/tutego/insel/lang/SetProperty.java, main()`

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

System.out.println( logProperty.isPresent() );           // true
usernameProperty.ifPresent( System.out::println );      // Chris
sizeProperty.map( Integer::parseInt ).ifPresent( System.out::println ); // 100
System.out.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 `logProperty` 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. Die Rückgabe ist `false`, wenn der Wert der Systemeigenschaft `"false"` ist, wenn er nicht existiert oder `null` ist.

16.4.5 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 der Programmierer unterscheidet sich die Anzahl der Zeichen für den Zeilenumbruch auf den bekannten Architekturen:

- ▶ 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 Zeilenumbruchzeichenfolge 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 (bzw. genau genommen eine mögliche Zeichenfolge) 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 Zeilenumbruchzeichenfolge.

16.4.6 Umgebungsvariablen des Betriebssystems

Fast jedes Betriebssystem nutzt das Konzept der *Umgebungsvariablen* (engl. *environment variables*); bekannt ist etwa `PATH` für den Suchpfad für Applikationen unter Windows und

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

unter Unix. Java macht es möglich, auf diese System-Umgebungsvariablen zuzugreifen. Dazu dienen zwei statische Methoden:

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

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

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

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

Tabelle 16.5 Auswahl einiger unter Windows verfügbarer Umgebungsvariablen

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



Beispiel

Gib die Umgebungsvariablen des Systems aus:

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

16.5 Sprachen der Länder

Beginnen Entwickler mit Ausgaben auf der Konsole oder grafischen Oberfläche, so verdrahten sie oft die Ausgabe fest mit einer Landessprache. Ändert sich die Sprache, kann die Software nicht mit anderen landesüblichen Regeln etwa bei der Formatierung von Fließkommazahlen 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:

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

- ▶ 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:

```
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 sind 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" );
```



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

- `Locale(String language)`
Erzeugt ein neues `Locale`-Objekt für die Sprache (`language`), die nach dem ISO-639-Standard gegeben ist. Ungültige Kennungen werden nicht erkannt.
- `Locale(String language, String country)`
Erzeugt ein `Locale`-Objekt für eine Sprache (`language`) nach ISO 639 und ein Land (`country`) nach dem ISO-3166-Standard.
- `Locale(String language, String country, String variant)`
Erzeugt ein `Locale`-Objekt für eine Sprache, ein Land und eine Variante. `variant` ist eine herstellerabhängige Angabe wie »WIN« oder »MAC«.

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

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

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

Die `Locale`-Klasse hat weitere Methoden; Entwickler sollten für den `Builder`, für `forLanguage-Tag(...)` und die neuen Erweiterungen und Filtermethoden die Javadoc studieren.⁷

Konstanten für einige Sprachen

Die `Locale`-Klasse besitzt Konstanten für häufig auftretende 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`.

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 von Methoden, die `Locale`-Objekte annehmen und anhand deren ihr Verhalten anpassen. Beispiele sind `printf(Locale, ...)`, `format(Locale, ...)` und `toLowerCase(Locale)`.



Tipp

Gibt es keine Variante einer Formatierungs- bzw. Parse-Methode mit einem `Locale`-Objekt, so 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 Fließkommazahl 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

Gib für Sprachen in ausgewählten Ländern zugängliche `Locale`-Informationen aus. Die Objekte `System.out` und `Locale.*` sind statisch importiert:

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

Listing 16.3 src/main/java/com/tutego/insel/locale/GermanyLocal.java, main()

```

out.println(GERMANY.getCountry());           // DE
out.println(GERMANY.getLanguage());          // de
out.println(GERMANY.getVariant());           //
out.println(GERMANY.getISO3Country());       // DEU
out.println(GERMANY.getISO3Language());      // deu
out.println(CANADA.getDisplayCountry());     // Kanada
out.println(GERMANY.getDisplayLanguage());   // Deutsch
out.println(GERMANY.getDisplayName());       // Deutsch (Deutschland)
out.println(CANADA.getDisplayName());        // Englisch (Kanada)
out.println(GERMANY.getDisplayName(FRENCH)); // allemand (Allemagne)
out.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 IsoCountry-Code bestimmt: PART1_ALPHA2 liefert den Code aus zwei Buchstaben, PART1_ALPHA3 aus drei Buchstaben, PART3 aus vier Buchstaben.

Auf der anderen Seite 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.

Die genannten Methoden liefern zwar Kürzel, aber sie sind nicht gedacht als für Menschen lesbare Ausgabe. 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

Weil Datumsberechnungen verschlungene Gebilde sind, können wir den Entwicklern von Java dankbar sein, dass sie uns viele Klassen zur Datumsberechnung und -formatierung zur Verfügung stellen. Die Entwickler haben die Klassen so abstrakt gehalten, dass lokale Besonderheiten wie Ausgabeformatierung, Parsen, Zeitzonen oder Sommer- und Winterzeit in verschiedenen Kalendern möglich sind.

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 1.1.1970 war ein Donnerstag mit wegweisenden Änderungen: Die Briten freuten sich, dass die Volljährigkeit von 24 Jahren auf 18 Jahre fiel, und wie in jedem Jahr wurde das Beschneidungsfest gefeiert. Für uns ist aber eine technische Neuerung von Belang: Der 1.1.1970, 0:00:00 UTC heißt auch *Unix-Epoche*, und eine *Unixzeit* wird relativ zu diesem Zeitpunkt in Sekunden beschrieben. So kommen wir 100.000.000 Sekunden nach dem 1.1.1970 beim 3. März 1973 um 09:46:40 aus. Das *Unix Billennium* wurde bei 1.000.000.000 Sekunden nach dem 1.1.1970 gefeiert und repräsentiert den 9. September 2001, 01:46:40.

16.6.2 `System.currentTimeMillis()`

Auch für uns Java-Entwickler ist die Unixzeit von Bedeutung, denn viele Zeiten in Java sind relativ zu diesem Datum. Der Zeitstempel 0 bezieht sich auf den 1.1.1970 0:00:00 Uhr UTC (Coordinated Universal Time) – das entspricht 1 Uhr nachts deutscher Zeit. Die Methode `System.currentTimeMillis()` liefert die vergangenen Millisekunden – nicht Sekunden! – relativ zum 1.1.1970, 00:00 Uhr UTC, wobei allerdings die Uhr des Betriebssystems nicht so genau gehen muss. Die Anzahl der Millisekunden wird in einem `long` repräsentiert, also in 64 Bit. Das reicht für etwa 300 Millionen Jahre.

16.6.3 Einfache Zeitumrechnungen durch TimeUnit

Eine Zeitdauer wird in Java oft durch Millisekunden ausgedrückt. 1.000 Millisekunden entsprechen 1 Sekunde, 1.000×60 Millisekunden 1 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 Aufzählungselemente 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;
System.out.println( TimeUnit.MILLISECONDS.toHours( v ) ); // 6
System.out.println( TimeUnit.HOURS.convert( v, TimeUnit.MILLISECONDS ) ); // 6
```

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

- `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, `SECONDS`, `MINUTES`, `HOURS`, `DAYS`

Aufzählungselemente 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 Zeitzonenwechsel seit 1970 dokumentiert.

Erster Überblick

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

Typ	Beschreibung	Feld(er)
<code>LocalDate</code>	Repräsentiert ein übliches Datum.	Jahr, Monat, Tag
<code>LocalTime</code>	Repräsentiert eine übliche Zeit.	Stunden, Minuten, Sekunden, Nanosekunden
<code>LocalDateTime</code>	Kombination aus Datum und Zeit	Jahr, Monat, Tag, Stunden, Minuten, Sekunden, Nanosekunden
<code>Period</code>	Dauer zwischen zwei <code>LocalDates</code>	Jahr, Monat, Tag
<code>Year</code>	nur Jahr	Jahr
<code>Month</code>	nur Monat	Monat
<code>MonthDay</code>	nur Monat und Tag	Monat, Tag
<code>OffsetTime</code>	Zeit mit Zeitzone	Stunden, Minuten, Sekunden, Nanosekunden, Zonen-Offset
<code>OffsetDateTime</code>	Datum und Zeit mit Zeitzone als UTC-Offset	Jahr, Monat, Tag, Stunden, Minuten, Sekunden, Nanosekunden, Zonen-Offset

Tabelle 16.6 Alle temporalen Klassen aus »`java.time`«

Typ	Beschreibung	Feld(er)
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 16.6 Alle temporalen Klassen aus »java.time« (Forts.)

16.7.1 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 im Nanosekundenbereich hat, *Maschinenzeit*. Die Maschinenzeit startet dabei von einer Zeit, die wir *Epoche* nennen, z. B. die Unix-Epoche.

Aus Abschnitt 7.2.5, »Sehen Kinder alles? Die Sichtbarkeit protected«, lässt sich gut ablesen, dass die meisten Klassen für uns Menschen gemacht sind und dass sich nur *Instant/Duration* auf die Maschinenzeit bezieht. *LocalDate*, *LocalTime* und *LocalDateTime* repräsentieren Menschenzeit ohne Bezug zu einer Zeitzone, *ZonedDateTime* mit Bezug auf eine Zeitzone. Bei der Auswahl der richtigen Zeitklassen für eine Aufgabenstellung 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 begann und endet; hier ist *Duration* korrekt, nicht *Period*.



Beispiel

```

    LocalDate now = LocalDate.now();
    System.out.println( now ); // z. B. 2024-01-31
    System.out.printf( "%d. %s %d%n",
                       now.getDayOfMonth(), now.getMonth(), now.getYear() );
    // z. B. 31. JANUARY 2024
    LocalDate bdayMLKing = LocalDate.of( 1929, Month.JANUARY, 15 );
    DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDate( Format-
    Style.MEDIUM );
    System.out.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();
System.out.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 Datum- und Zeit, 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 immutablen 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.

Tabelle 16.7 Namensmuster in der Date-Time-API

Methode	Klassen-/ Exemplarmethode	grundsätzliche Bedeutung
*Into()	Exemplar	Kombiniert ein eigenes Objekt mit einem anderen Zielobjekt.

Tabelle 16.7 Namensmuster in der Date-Time-API (Forts.)

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.

16.7.2 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(...)`-Fabrikmethoden 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. Weiterhin gibt es `long toEpochDay()` und `long toEpochSecond(LocalDate 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 `minusYear(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 `LocaleDate` 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 endDateExclusive )`
- ▶ `Stream<LocalDate> datesUntil( LocalDate endDateExclusive, Period step )`

16.8 Logging mit Java

Das Loggen (Protokollieren) von Informationen über Programmmzustände 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

Bei den Logging-Bibliotheken und APIs ist die Java-Welt leider gespalten. Da die Java-Standardbibliothek in den ersten Versionen keine Logging-API anbot, füllte die Open-Source-Bibliothek *Log4j* schnell diese Lücke. Sie wird heute in nahezu jedem größeren Java-Projekt eingesetzt. Als in Java 1.4 die Logging-API (JSR 47) einzog, war die Java-Gemeinde erstaunt, dass `java.util.logging` (*JUL*) weder API-kompatibel mit dem beliebten *Log4j* noch so leistungsfähig wie *Log4j* ist.⁸

Im Laufe der Jahre veränderte sich das Bild. Während in der Anfangszeit Entwickler ausschließlich auf *Log4j* bauten, werden es langsam mehr Projekte mit der JUL. Ein erster Grund ist der, dass einige Entwickler externe Abhängigkeiten vermeiden wollen (wobei das nicht wirklich funktioniert, da nahezu jede eingebundene Java-Bibliothek selbst auf *Log4j* baut). Der zweite Grund ist, dass für viele Projekte JUL einfach reicht. In der Praxis bedeutet dies für größere Projekte, dass mehrere Logging-Konfigurationen das eigene Programm verschmutzen, da jede Logging-Implementierung unterschiedlich konfiguriert wird.

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:

⁸ Die Standard-Logging-API ist dagegen nur ein laues Lüftchen, das nur Grundlegendes wie hierarchische Logger bietet. An die Leistungsfähigkeit von *Log4j* mit einer großen Anzahl von Schreibern in Dateien, Syslog-/NT-Logger, Datenbanken, Versand über das Netzwerk kommt das Standard-Logging nicht heran.

Listing 16.4 src/main/java/com/tutego/insel/logging/JULDemo.java, JULDemo

```

package com.tutego.insel.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 JULDemo {

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

    public static void main( String[] args ) {
        Instant start = now();
        log.info( "About to start" );

        try {
            log.log( Level.INFO, "Lets 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:

```

Jan. 24, 2024 7:47:46 PM com.tutego.insel.logging.JULDemo main
INFO: About to start
Jan. 24, 2024 7:47:46 PM com.tutego.insel.logging.JULDemo main
INFO: Lets try to throw null
Jan. 24, 2024 7:47:46 PM com.tutego.insel.logging.JULDemo main
SEVERE: Oh Oh
java.lang.NullPointerException: Cannot throw exception because "null" is null
    at com.tutego.insel.logging.JULDemo.main(JULDemo.java:20)

Jan. 24, 2024 7:47:46 PM com.tutego.insel.logging.JULDemo main
INFO: Runtime: 35 ms

```

Das Logger-Objekt

Zentral ist das `Logger`-Objekt, das über `Logger.getAnonymousLogger()` oder über `Logger.getLogger(String name)` geholt werden kann, wobei `name` in der Regel mit dem voll qualifizierten Klassennamen belegt ist. 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:⁹

- ▶ `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
<code>SEVERE</code>	<code>log(Level.SEVERE, msg)</code>	<code>severe(String msg)</code>
<code>WARNING</code>	<code>log(Level.WARNING, msg)</code>	<code>warning(String msg)</code>
<code>INFO</code>	<code>log(Level.INFO, msg)</code>	<code>info(String msg)</code>
<code>CONFIG</code>	<code>log(Level.CONFIG, msg)</code>	<code>config(String msg)</code>
<code>FINE</code>	<code>log(Level.FINE, msg)</code>	<code>fine(String msg)</code>
<code>FINER</code>	<code>log(Level.FINER, msg)</code>	<code>finer(String msg)</code>
<code>FINEST</code>	<code>log(Level.FINEST, msg)</code>	<code>finest(String msg)</code>

Tabelle 16.8 Log-Level und Methoden

⁹ 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.

Alle diese Methoden setzen eine Mitteilung vom Typ `String` ab. Sollen eine Ausnahme und der dazugehörige Stack-Trace geloggt werden, müssen Entwickler zu folgender Logger-Methode greifen, 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 das Fettgedruckte für die Abhängigkeit:

Listing 16.5 pom.xml

```
<project ...>
...
<properties>
  <maven.compiler.target>21</maven.compiler.target>
  <maven.compiler.source>21</maven.compiler.source>
  <maven.compiler.release>21</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 16.6 `src/main/java/SparkServer.java`

```
public class SparkServer {
    public static void main( String[] args ) {
        spark.Spark.get( "/hello", ( req, res ) -> "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/`.



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

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

Ein Maven-Build besteht aus einem dreistufigen Lebenszyklus `clean`, `default` und `site`. Innerhalb dessen 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, wo 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 Aufbau und 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 unangenehm. Java-Entwickler sollten für den kommerziellen Vertrieb ihr Augenmerk verstärkt auf Software unter der BSD-Lizenz (die Apache-Lizenz gehört in diese Gruppe) und unter der LGPL-Lizenz richten. Die Apache-Gruppe hat mit den *Apache Commons* (<http://commons.apache.org>) eine hübsche Sammlung an Klassen und Methoden zusammengetragen, 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 Suche über bestimmte Stichwörter durch mehr als 1 Milliarde Quellcodezeilen verschiedener Programmiersprachen ermöglicht; erstaunlich, wie viele Entwickler »F*ck« schreiben. Und »Porn Groove« kannte ich vor dieser Suche auch noch nicht.

Inhalt

Materialien zum Buch	31
Vorwort	33
 1 Java ist auch eine Sprache	 49
1.1 Historischer Hintergrund	50
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	52
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	58
1.2.11 Das Angebot an Bibliotheken und Werkzeugen	59
1.2.12 Vergleichbar einfache Syntax	60
1.2.13 Verzicht auf umstrittene Konzepte	61
1.2.14 Java ist Open Source	62
1.2.15 Wofür sich Java weniger eignet	62
1.3 Java im Vergleich zu anderen Sprachen *	63
1.3.1 Java und C(++)	64
1.3.2 Java und JavaScript	64
1.3.3 Ein Wort zu Microsoft, Java und zu J++	65
1.3.4 Java und C#/.NET	66
1.4 Weiterentwicklung und Verluste	66
1.4.1 Die Entwicklung von Java und seine Zukunftsaussichten	67
1.4.2 Features, Enhancements (Erweiterungen) und ein JSR	68
1.4.3 Applets	68
1.4.4 JavaFX	69
1.5 Java-Plattformen: Java SE, Jakarta EE, Java ME, Java Card	70
1.5.1 Die Java SE-Plattform	70
1.5.2 Java ME: Java für die Kleinen	72

1.5.3	Java für die ganz, ganz Kleinen	73
1.5.4	Java für die Großen: Jakarta EE (ehemals Java EE)	73
1.6	Java SE-Implementierungen	74
1.6.1	OpenJDK	74
1.6.2	Oracle JDK	75
1.7	JDK installieren	77
1.7.1	Oracle JDK unter Windows installieren	77
1.8	Das erste Programm compilieren und testen	79
1.8.1	99 Flaschen Bier an der Wand: Der Countdown geht weiter!	79
1.8.2	Der Compilerlauf	80
1.8.3	Die Laufzeitumgebung	81
1.8.4	Häufige Compiler- und Interpreter-Probleme	82
1.9	Entwicklungsumgebungen	83
1.9.1	IntelliJ IDEA	83
1.9.2	Eclipse IDE	87
1.9.3	NetBeans	89
1.10	Zum Weiterlesen	89

2 Imperative Sprachkonzepte 91

2.1	Elemente der Programmiersprache Java	91
2.1.1	Token	92
2.1.2	Textkodierung durch Unicode-Zeichen	93
2.1.3	Bezeichner	93
2.1.4	Literale	95
2.1.5	(Reservierte) Schlüsselwörter	95
2.1.6	Zusammenfassung der lexikalischen Analyse	97
2.1.7	Kommentare	98
2.2	Von der Klasse zur Anweisung	99
2.2.1	Was sind Anweisungen?	100
2.2.2	Klassendeklaration	100
2.2.3	Die Reise beginnt am main(String[])	101
2.2.4	Der erste Methodenaufruf: println(...)	102
2.2.5	Atomare Anweisungen und Anweisungssequenzen	103
2.2.6	Mehr zu print(...), println(...) und printf(...) für Bildschirmausgaben	104
2.2.7	Die API-Dokumentation	106
2.2.8	Ausdrücke	107
2.2.9	Ausdrucksanweisung	107

2.2.10	Erster Einblick in die Objektorientierung	108
2.2.11	Modifizierer	109
2.2.12	Gruppieren von Anweisungen mit Blöcken	109
2.3	Datentypen, Typisierung, Variablen und Zuweisungen	111
2.3.1	Primitive Datentypen im Überblick	112
2.3.2	Variablendeklarationen	115
2.3.3	Automatisches Feststellen der Typen mit var	117
2.3.4	Finale Variablen und der Modifizierer final	118
2.3.5	Konsoleneingaben	119
2.3.6	Wahrheitswerte	121
2.3.7	Ganzzahlige Datentypen	121
2.3.8	Unterstriche in Zahlen	123
2.3.9	Alphanumerische Zeichen	124
2.3.10	Fließkommazahlen mit den Datentypen float und double	124
2.3.11	Gute Namen, schlechte Namen	126
2.3.12	Keine automatische Initialisierung von lokalen Variablen	127
2.4	Ausdrücke, Operanden und Operatoren	127
2.4.1	Zuweisungsoperator	128
2.4.2	Arithmetische Operatoren	130
2.4.3	Unäres Minus und Plus	133
2.4.4	Präfix- oder Postfix-Inkrement und -Dekrement	133
2.4.5	Zuweisung mit Operation (Verbundoperator)	135
2.4.6	Die relationalen Operatoren und die Gleichheitsoperatoren	137
2.4.7	Logische Operatoren: Nicht, Und, Oder, XOR	139
2.4.8	Kurzschluss-Operatoren	140
2.4.9	Der Rang der Operatoren in der Auswertungsreihenfolge	141
2.4.10	Die Typumwandlung (das Casting)	144
2.4.11	Überladenes Plus für Strings	150
2.4.12	Operator vermisst *	151
2.5	Bedingte Anweisungen oder Fallunterscheidungen	151
2.5.1	Verzweigung mit der if-Anweisung	152
2.5.2	Die Alternative mit einer if-else-Anweisung wählen	154
2.5.3	Der Bedingungsoperator	158
2.5.4	Die switch-Anweisung bietet die Alternative	161
2.5.5	Switch-Ausdrücke	168
2.6	Immer das Gleiche mit den Schleifen	170
2.6.1	Die while-Schleife	171
2.6.2	Die do-while-Schleife	173
2.6.3	Die for-Schleife	175
2.6.4	Schleifenbedingungen und Vergleiche mit == *	179

2.6.5	Schleifenabbruch mit break und zurück zum Test mit continue	181
2.6.6	break und continue mit Marken *	184
2.7	Methoden einer Klasse	188
2.7.1	Bestandteile einer Methode	188
2.7.2	Signatur-Beschreibung in der Java-API	190
2.7.3	Aufruf einer Methode	191
2.7.4	Methoden ohne Parameter deklarieren	192
2.7.5	Statische Methoden (Klassenmethoden)	193
2.7.6	Parameter, Argument und Wertübergabe	194
2.7.7	Methoden vorzeitig mit return beenden	196
2.7.8	Methoden mit Rückgaben	197
2.7.9	Methoden überladen	202
2.7.10	Gültigkeitsbereich	204
2.7.11	Vorgegebener Wert für nicht aufgeführte Argumente *	206
2.7.12	Rekursive Methoden *	206
2.7.13	Die Türme von Hanoi *	210
2.8	Zum Weiterlesen	212
3	Klassen und Objekte	213
3.1	Objektorientierte Programmierung (OOP)	213
3.1.1	Warum überhaupt OOP?	214
3.1.2	Denk ich an Java, denk ich an Wiederverwendbarkeit	214
3.2	Eigenschaften einer Klasse	215
3.2.1	Klassenarbeit mit Point	216
3.3	Natürlich modellieren mit der UML (Unified Modeling Language) *	217
3.3.1	Wichtige Diagrammtypen der UML *	217
3.4	Neue Objekte erzeugen	219
3.4.1	Ein Exemplar einer Klasse mit dem Schlüsselwort new anlegen	219
3.4.2	Deklarieren von Referenzvariablen	219
3.4.3	Jetzt mach mal 'nen Punkt: Zugriff auf Objektvariablen und -methoden	220
3.4.4	Der Zusammenhang von new, Heap und Garbage-Collector	225
3.4.5	Überblick über Point-Methoden	226
3.4.6	Konstruktoren nutzen	230
3.5	ZZZZZnake	231
3.6	Pakete schnüren, Importe und Compilationseinheiten	233
3.6.1	Java-Pakete	234

3.6.2	Pakete der Standardbibliothek	234
3.6.3	Volle Qualifizierung und import-Deklaration	234
3.6.4	Mit import p1.p2.* alle Typen eines Pakets erreichen	236
3.6.5	Hierarchische Strukturen über Pakete und die Spiegelung im Dateisystem ..	237
3.6.6	Die package-Deklaration	237
3.6.7	Unbenanntes Paket (default package)	239
3.6.8	Compilationseinheit (Compilation Unit)	239
3.6.9	Statischer Import *	240
3.7	Mit Referenzen arbeiten, Vielfalt, Identität, Gleichwertigkeit	241
3.7.1	null-Referenz und die Frage der Philosophie	241
3.7.2	Alles auf null? Referenzen testen	243
3.7.3	Zuweisungen bei Referenzen	244
3.7.4	Methoden mit Referenztypen als Parameter	246
3.7.5	Identität von Objekten	250
3.7.6	Gleichwertigkeit und die Methode equals(...)	251
3.8	Zum Weiterlesen	253

4 Arrays und ihre Anwendungen 255

4.1	Einfache Feldarbeit	255
4.1.1	Grundbestandteile	256
4.1.2	Deklaration von Array-Variablen	257
4.1.3	Array-Objekte mit new erzeugen	258
4.1.4	Arrays mit { Inhalt }	259
4.1.5	Die Länge eines Arrays über die Objektvariable length auslesen	260
4.1.6	Zugriff auf die Elemente über den Index	260
4.1.7	Typische Array-Fehler	262
4.1.8	Arrays an Methoden übergeben	264
4.1.9	Mehrere Rückgabewerte *	264
4.1.10	Vorinitialisierte Arrays	265
4.2	Die erweiterte for-Schleife	267
4.3	Methode mit variabler Argumentanzahl (Varargs)	271
4.3.1	System.out.printf(...) nimmt eine beliebige Anzahl von Argumenten an	271
4.3.2	Durchschnitt finden von variablen Argumenten	272
4.3.3	Varargs-Designtipps *	273
4.4	Mehrdimensionale Arrays *	273
4.4.1	Nichtrechteckige Arrays *	276

4.5	Bibliotheksunterstützung von Arrays	279
4.5.1	Klonen kann sich lohnen – Arrays vermehren	279
4.5.2	Warum »können« Arrays so wenig?	280
4.5.3	Array-Inhalte kopieren	280
4.6	Die Klasse Arrays zum Vergleichen, Füllen, Suchen und Sortieren nutzen	282
4.6.1	Eine lange Schlange	292
4.7	Der Einstiegspunkt für das Laufzeitsystem: main(...)	295
4.7.1	Korrekte Deklaration der Startmethode	295
4.7.2	Kommandozeilenargumente verarbeiten	296
4.7.3	Der Rückgabetypp von main(...) und System.exit(int) *	297
4.8	Zum Weiterlesen	298

5 Der Umgang mit Zeichen und Zeichenketten

5.1	Von ASCII über ISO-8859-1 zu Unicode	299
5.1.1	ASCII	299
5.1.2	ISO/IEC 8859-1	300
5.1.3	Unicode	301
5.1.4	Unicode-Zeichenkodierung	303
5.1.5	Escape-Sequenzen/Fluchtsymbole	304
5.1.6	Schreibweise für Unicode-Zeichen und Unicode-Escapes	305
5.1.7	Java-Versionen gehen mit dem Unicode-Standard Hand in Hand *	306
5.2	Datentypen für Zeichen und Zeichenfolgen	307
5.3	Die Character-Klasse	308
5.3.1	Ist das so?	308
5.3.2	Zeichen in Großbuchstaben/Kleinbuchstaben konvertieren	311
5.3.3	Vom Zeichen zum String	312
5.3.4	Von char in int: vom Zeichen zur Zahl *	312
5.4	Zeichenfolgen	313
5.5	Die Klasse String und ihre Methoden	315
5.5.1	String-Literale als String-Objekte für konstante Zeichenketten	316
5.5.2	Konkatenation mit +	316
5.5.3	Mehrzeilige Textblöcke mit ""	317
5.5.4	String-Länge und Test auf Leer-String	322
5.5.5	Zugriff auf ein bestimmtes Zeichen mit charAt(int)	323
5.5.6	Nach enthaltenen Zeichen und Zeichenfolgen suchen	324
5.5.7	Das Hangman-Spiel	327

5.5.8	Gut, dass wir verglichen haben	329
5.5.9	String-Teile extrahieren	334
5.5.10	Strings anhängen, zusammenfügen, Groß-/Kleinschreibung und Weißraum	338
5.5.11	Gesucht, gefunden, ersetzt	342
5.5.12	String-Objekte mit Konstruktoren und aus Wiederholungen erzeugen *	344
5.6	Veränderbare Zeichenketten mit StringBuilder und StringBuffer	348
5.6.1	Anlegen von StringBuilder-Objekten	348
5.6.2	StringBuilder in andere Zeichenkettenformate konvertieren	349
5.6.3	Zeichen(folgen) erfragen	349
5.6.4	Daten anhängen	350
5.6.5	Zeichen(folgen) setzen, löschen und umdrehen	351
5.6.6	Länge und Kapazität eines StringBuilder-Objekts *	353
5.6.7	Vergleich von StringBuilder-Exemplaren und Strings mit StringBuilder	355
5.6.8	hashCode() bei StringBuilder *	356
5.7	CharSequence als Basistyp	357
5.8	Konvertieren zwischen Primitiven und Strings	360
5.8.1	Unterschiedliche Typen in String-Repräsentationen konvertieren	360
5.8.2	String-Inhalt in einen primitiven Wert konvertieren	362
5.8.3	String-Repräsentation in den Formaten Binär, Hex und Oktal *	364
5.9	Strings zusammenhängen (konkatenieren)	368
5.9.1	Strings mit StringJoiner zusammenhängen	368
5.10	Zerlegen von Zeichenketten	370
5.10.1	Splitten von Zeichenketten mit split(...)	371
5.10.2	Yes we can, yes we scan – die Klasse Scanner	372
5.11	Ausgaben formatieren	375
5.11.1	Formatieren und Ausgeben mit format()	375
5.12	Zum Weiterlesen	381
6	Eigene Klassen schreiben	383
6.1	Eigene Klassen mit Eigenschaften deklarieren	383
6.1.1	Objektvariablen deklarieren	384
6.1.2	Methoden deklarieren	387
6.1.3	Verdeckte (shadowed) Variablen	389
6.1.4	Die this-Referenz	391

6.2	Privatsphäre und Sichtbarkeit	394
6.2.1	Für die Öffentlichkeit: public	395
6.2.2	Kein Public Viewing – Passwörter sind privat	395
6.2.3	Wieso nicht freie Methoden und Variablen für alle?	397
6.2.4	Privat ist nicht ganz privat: Es kommt darauf an, wer's sieht *	397
6.2.5	Zugriffsmethoden für Objektvariablen deklarieren	398
6.2.6	Setter und Getter nach der JavaBeans-Spezifikation	398
6.2.7	Paketsichtbar	401
6.2.8	Zusammenfassung zur Sichtbarkeit	402
6.3	Eine für alle – statische Methoden und Klassenvariablen	405
6.3.1	Warum statische Eigenschaften sinnvoll sind	406
6.3.2	Statische Eigenschaften mit static	406
6.3.3	Statische Eigenschaften über Referenzen nutzen? *	408
6.3.4	Warum die Groß- und Kleinschreibung wichtig ist *	409
6.3.5	Statische Variablen zum Datenaustausch *	409
6.3.6	Statische Eigenschaften und Objekteigenschaften *	411
6.4	Konstanten und Aufzählungen	412
6.4.1	Konstanten über statische finale Variablen	412
6.4.2	Typunsichere Aufzählungen	413
6.4.3	Aufzählungstypen: typsichere Aufzählungen mit enum	415
6.5	Objekte anlegen und zerstören	420
6.5.1	Konstruktoren schreiben	420
6.5.2	Verwandtschaft von Methode und Konstruktor	422
6.5.3	Der Standard-Konstruktor (default constructor)	423
6.5.4	Parametrisierte und überladene Konstruktoren	424
6.5.5	Copy-Konstruktor	426
6.5.6	Einen anderen Konstruktor der gleichen Klasse mit this(...) aufrufen	427
6.5.7	Immutable-Objekte und Wither-Methoden	430
6.5.8	Ihr fehlt uns nicht – der Garbage-Collector	432
6.6	Klassen- und Objektinitialisierung *	434
6.6.1	Initialisierung von Objektvariablen	434
6.6.2	Statische Blöcke als Klasseninitialisierer	436
6.6.3	Initialisierung von Klassenvariablen	437
6.6.4	Eincompilierte Belegungen der Klassenvariablen	437
6.6.5	Exemplarinitialisierer (Instanzinitialisierer)	438
6.6.6	Finale Werte im Konstruktor und in statischen Blöcken setzen	441
6.7	Zum Weiterlesen	443

7	Objektorientierte Beziehungsfragen	445
7.1	Assoziationen zwischen Objekten	445
7.1.1	Unidirektionale 1:1-Beziehung	446
7.1.2	Zwei Freunde müsst ihr werden – bidirektionale 1:1-Beziehungen	448
7.1.3	Unidirektionale 1:n-Beziehung	449
7.2	Vererbung	455
7.2.1	Vererbung in Java	456
7.2.2	Ereignisse modellieren	456
7.2.3	Die implizite Basisklasse java.lang.Object	458
7.2.4	Einfach- und Mehrfachvererbung *	459
7.2.5	Sehen Kinder alles? Die Sichtbarkeit protected	459
7.2.6	Konstruktoren in der Vererbung und super(...)	460
7.3	Typen in Hierarchien	465
7.3.1	Automatische und explizite Typumwandlung	466
7.3.2	Das Substitutionsprinzip	469
7.3.3	Typen mit dem instanceof-Operator testen	471
7.3.4	Pattern-Matching bei instanceof	474
7.3.5	Pattern-Matching bei switch	476
7.4	Methoden überschreiben	479
7.4.1	Methoden in Unterklassen mit neuem Verhalten ausstatten	479
7.4.2	Mit super an die Eltern	483
7.5	Drum prüfe, wer sich dynamisch bindet	485
7.5.1	Gebunden an toString()	486
7.5.2	Implementierung von System.out.println(Object)	487
7.6	Finale Klassen und finale Methoden	488
7.6.1	Finale Klassen	488
7.6.2	Nicht überschreibbare (finale) Methoden	489
7.7	Abstrakte Klassen und abstrakte Methoden	490
7.7.1	Abstrakte Klassen	491
7.7.2	Abstrakte Methoden	492
7.8	Weiteres zum Überschreiben und dynamischen Binden	499
7.8.1	Nicht dynamisch gebunden bei privaten, statischen und finalen Methoden	499
7.8.2	Kovariante Rückgabetypen	500
7.8.3	Array-Typen und Kovarianz *	501
7.8.4	Dynamisch gebunden auch bei Konstruktoraufrufen *	502
7.8.5	Keine dynamische Bindung bei überdeckten Objektvariablen *	504
7.9	Zum Weiterlesen und Programmieraufgabe	505

8	Records, Schnittstellen, Aufzählungen, versiegelte Klassen	507
8.1	Records	507
8.1.1	Einfache Records	507
8.1.2	Records mit Methoden	509
8.1.3	Konstruktoren von Records anpassen	511
8.1.4	Konstruktoren ergänzen	513
8.1.5	Record-Patterns zur Destrukturierung	514
8.1.6	Records: Zusammenfassung	518
8.2	Schnittstellen	518
8.2.1	Schnittstellen sind neue Typen	518
8.2.2	Schnittstellen deklarieren	519
8.2.3	Abstrakte Methoden in Schnittstellen	519
8.2.4	Implementieren von Schnittstellen	520
8.2.5	Ein Polymorphie-Beispiel mit Schnittstellen	523
8.2.6	Records implementieren Schnittstellen	525
8.2.7	Die Mehrfachvererbung bei Schnittstellen	526
8.2.8	Keine Kollisionsgefahr bei Mehrfachvererbung *	529
8.2.9	Erweitern von Interfaces – Subinterfaces	530
8.2.10	Konstantendeklarationen bei Schnittstellen	531
8.2.11	Nachträgliches Implementieren von Schnittstellen *	531
8.2.12	Statische ausprogrammierte Methoden in Schnittstellen	532
8.2.13	Erweitern und Ändern von Schnittstellen	534
8.2.14	Default-Methoden	536
8.2.15	Erweiterte Schnittstellen deklarieren und nutzen	537
8.2.16	Öffentliche und private Schnittstellenmethoden	541
8.2.17	Erweiterte Schnittstellen, Mehrfachvererbung und Mehrdeutigkeiten *	541
8.2.18	Bausteine bilden mit Default-Methoden *	545
8.2.19	Markierungsschnittstellen *	548
8.2.20	(Abstrakte) Klassen und Schnittstellen im Vergleich	549
8.3	Aufzählungstypen	550
8.3.1	Methoden auf Enum-Objekten	550
8.3.2	Aufzählungen mit eigenen Methoden *	554
8.3.3	enum mit eigenen Konstruktoren und Initialisierern	556
8.3.4	enum kann Schnittstellen implementieren	559
8.4	Versiegelte Klassen und Schnittstellen	560
8.4.1	Versiegelte Klassen und Schnittstellen (sealed classes/interfaces)	562
8.4.2	Unterklassen sind final, sealed, non-sealed	564
8.4.3	Vollständige Abdeckung von Aufzählungen	564

8.4.4	Abkürzende Schreibweisen	565
8.4.5	Versiegelte Schnittstellen und Records	566
8.5	Zum Weiterlesen	568
9	Ausnahmen müssen sein	569
9.1	Problembereiche einzäunen	570
9.1.1	Exceptions in Java mit try und catch	570
9.1.2	Geprüfte und ungeprüfte Ausnahmen	571
9.1.3	Eine NumberFormatException fliegt (ungeprüfte Ausnahme)	571
9.1.4	Datum-Zeitstempel in Textdatei anhängen (geprüfte Ausnahme)	573
9.1.5	Wiederholung abgebrochener Bereiche *	576
9.1.6	Bitte nicht schlucken – leere catch-Blöcke	576
9.1.7	Mehrere Ausnahmen auffangen	577
9.1.8	Zusammenfassen gleicher catch-Blöcke mit dem multi-catch	578
9.2	Ausnahmen mit throws nach oben reichen	579
9.2.1	throws bei Konstruktoren und Methoden	579
9.3	Die Klassenhierarchie der Ausnahmen	580
9.3.1	Eigenschaften des Exception-Objekts	580
9.3.2	Basistyp Throwable	581
9.3.3	Die Exception-Hierarchie	582
9.3.4	Oberausnahmen auffangen	583
9.3.5	Schon gefangen?	585
9.3.6	Ablauf einer Ausnahmesituation	586
9.3.7	Nicht zu allgemein fangen!	586
9.3.8	Bekannte RuntimeException-Klassen	588
9.3.9	Kann man abfangen, muss man aber nicht	589
9.4	Abschlussbehandlung mit finally	590
9.5	Auslösen eigener Exceptions	595
9.5.1	Mit throw Ausnahmen auslösen	595
9.5.2	Vorhandene Runtime-Ausnahmetypen kennen und nutzen	597
9.5.3	Parameter testen und gute Fehlermeldungen	600
9.5.4	Neue Exception-Klassen deklarieren	601
9.5.5	Eigene Ausnahmen als Unterklassen von Exception oder RuntimeException?	603
9.5.6	Ausnahmen abfangen und weiterleiten *	606
9.5.7	Aufruf-Stack von Ausnahmen verändern *	607
9.5.8	Geschachtelte Ausnahmen *	608

9.6	try mit Ressourcen (automatisches Ressourcen-Management)	611
9.6.1	try mit Ressourcen	612
9.6.2	Die Schnittstelle AutoCloseable	613
9.6.3	Ausnahmen vom close()	614
9.6.4	Typen, die AutoCloseable und Closeable sind	615
9.6.5	Mehrere Ressourcen nutzen	616
9.6.6	try mit Ressourcen auf null-Ressourcen	618
9.6.7	Unterdrückte Ausnahmen *	618
9.7	Besonderheiten bei der Ausnahmebehandlung *	621
9.7.1	Rückgabewerte bei ausgelösten Ausnahmen	622
9.7.2	Ausnahmen und Rückgaben verschwinden – das Duo return und finally	622
9.7.3	throws bei überschriebenen Methoden	623
9.7.4	Nicht erreichbare catch-Klauseln	625
9.8	Harte Fehler – Error *	627
9.9	Assertions *	627
9.9.1	Assertions in eigenen Programmen nutzen	628
9.9.2	Assertions aktivieren und Laufzeit-Errors	628
9.9.3	Assertions feiner aktivieren oder deaktivieren	630
9.10	Zum Weiterlesen	630

10 Geschachtelte Typen 631

10.1	Geschachtelte Klassen, Schnittstellen und Aufzählungen	631
10.2	Statische geschachtelte Typen	633
10.3	Nichtstatische geschachtelte Typen	635
10.3.1	Exemplare innerer Klassen erzeugen	635
10.3.2	Die this-Referenz	636
10.3.3	Vom Compiler generierte Klassendateien *	637
10.4	Lokale Klassen	638
10.4.1	Beispiel mit eigener Klassendeklaration	638
10.4.2	Lokale Klasse für einen Timer nutzen	639
10.5	Anonyme innere Klassen	639
10.5.1	Nutzung einer anonymen inneren Klasse für den Timer	640
10.5.2	Umsetzung innerer anonymer Klassen *	641
10.5.3	Konstruktoren innerer anonymer Klassen	642

10.6 Vermischtes	644
10.6.1 Zugriff auf lokale Variablen aus lokalen und anonymen Klassen *	644
10.6.2 this in Unterklassen *	644
10.6.3 Geschachtelte Klassen greifen auf private Eigenschaften zu	645
10.6.4 Nester	647
10.7 Zum Weiterlesen	648
 11 Besondere Typen der Java SE	 649
11.1 Object ist die Mutter aller Klassen	650
11.1.1 Klassenobjekte	650
11.1.2 Objektidentifikation mit toString()	651
11.1.3 Objektgleichwertigkeit mit equals(...) und Identität	653
11.1.4 Klonen eines Objekts mit clone() *	659
11.1.5 Hashwerte über hashCode() liefern *	664
11.1.6 System.identityHashCode(...) und das Problem der nicht eindeutigen Objektverweise *	671
11.1.7 Synchronisation *	672
11.2 Schwache Referenzen und Cleaner	672
11.3 Die Utility-Klasse java.util.Objects	673
11.3.1 Eingebaute null-Tests für equals(...)/hashCode()	674
11.3.2 Objects.toString(...)	675
11.3.3 null-Prüfungen mit eingebauter Ausnahmebehandlung	675
11.3.4 Tests auf null	676
11.3.5 Indexbezogene Programmargumente auf Korrektheit prüfen	677
11.4 Vergleichen von Objekten und Ordnung herstellen	677
11.4.1 Natürlich geordnet oder nicht?	678
11.4.2 compare*()-Methode der Schnittstellen Comparable und Comparator	679
11.4.3 Rückgabewerte kodieren die Ordnung	680
11.4.4 Mit einem Beispiel-Comparator Süßigkeiten nach Kalorien sortieren	680
11.4.5 Tipps für Comparator- und Comparable-Implementierungen	682
11.4.6 Statische und Default-Methoden in Comparator	683
11.5 Wrapper-Klassen und Autoboxing	686
11.5.1 Wrapper-Objekte erzeugen	688
11.5.2 Konvertierungen in eine String-Repräsentation	689
11.5.3 Von einer String-Repräsentation parsen	691

11.5.4	Die Basisklasse Number für numerische Wrapper-Objekte	691
11.5.5	Vergleiche durchführen mit compare*(...), compareTo(...), equals(...) und Hashwerten	693
11.5.6	Statische Reduzierungsmethoden in Wrapper-Klassen	696
11.5.7	Konstanten für die Größe eines primitiven Typs	697
11.5.8	Behandeln von vorzeichenlosen Zahlen *	697
11.5.9	Die Klassen Integer und Long	699
11.5.10	Die Klassen Double und Float für Fließkommazahlen	700
11.5.11	Die Boolean-Klasse	700
11.5.12	Autoboxing: Boxing und Unboxing	701
11.6	Iterator, Iterable *	706
11.6.1	Die Schnittstelle Iterator	706
11.6.2	Wer den Iterator liefert	710
11.6.3	Die Schnittstelle Iterable	710
11.6.4	Erweitertes for und Iterable	711
11.6.5	Interne Iteration	711
11.6.6	Ein eigenes Iterable implementieren *	712
11.7	Annotationen in der Java SE	714
11.7.1	Orte für Annotationen	714
11.7.2	Annotationstypen aus java.lang	715
11.7.3	@Deprecated	715
11.7.4	Annotationen mit zusätzlichen Informationen	716
11.7.5	@SuppressWarnings	716
11.8	Zum Weiterlesen	719
12	Generics<T>	721
12.1	Einführung in Java Generics	721
12.1.1	Mensch versus Maschine – Typprüfung des Compilers und der Laufzeitumgebung	721
12.1.2	Raketen	722
12.1.3	Generische Typen deklarieren	724
12.1.4	Generics nutzen	726
12.1.5	Diamonds are forever	729
12.1.6	Generische Schnittstellen	732
12.1.7	Generische Methoden/Konstruktoren und Typ-Inferenz	733
12.2	Umsetzen der Generics, Typlöschung und Raw-Types	737
12.2.1	Realisierungsmöglichkeiten	737
12.2.2	Typlöschung (Type Erasure)	738

12.2.3	Probleme der Typlöschung	739
12.2.4	Raw-Type	743
12.3	Die Typen über Bounds einschränken	745
12.3.1	Einfache Einschränkungen mit extends	745
12.3.2	Weitere Obertypen mit &	748
12.4	Typparameter in der throws-Klausel *	748
12.4.1	Deklaration einer Klasse mit Typvariable <E extends Exception>	748
12.4.2	Parametrisierter Typ bei Typvariable <E extends Exception>	749
12.5	Generics und Vererbung, Invarianz	752
12.5.1	Arrays sind kovariant	752
12.5.2	Generics sind nicht kovariant, sondern invariant	752
12.5.3	Wildcards mit ?	753
12.5.4	Bounded Wildcards	756
12.5.5	Bounded-Wildcard-Typen und Bounded-Typvariablen	759
12.5.6	Das LESS-Prinzip	761
12.5.7	Enum<E extends Enum<E>> *	764
12.6	Konsequenzen der Typlöschung: Typ-Token, Arrays und Brücken *	766
12.6.1	Typ-Token	766
12.6.2	Super-Type-Token	768
12.6.3	Generics und Arrays	769
12.6.4	Brückenmethoden *	770
12.7	Zum Weiterlesen	775
13	Lambda-Ausdrücke und funktionale Programmierung	777
13.1	Funktionale Schnittstellen und Lambda-Ausdrücke	777
13.1.1	Klassen implementieren Schnittstellen	777
13.1.2	Lambda-Ausdrücke implementieren Schnittstellen	779
13.1.3	Funktionale Schnittstellen	780
13.1.4	Der Typ eines Lambda-Ausdrucks ergibt sich durch den Zieltyp	781
13.1.5	Annotation @FunctionalInterface	786
13.1.6	Syntax für Lambda-Ausdrücke	787
13.1.7	Die Umgebung der Lambda-Ausdrücke und Variablenzugriffe	792
13.1.8	Ausnahmen in Lambda-Ausdrücken	797
13.1.9	Klassen mit einer abstrakten Methode als funktionale Schnittstelle? *	801
13.2	Methodenreferenz	801
13.2.1	Motivation	801
13.2.2	Methodenreferenzen mit ::	802

13.2.3	Varianten von Methodenreferenzen	803
13.3	Konstruktorreferenz	805
13.3.1	Konstruktorreferenzen schreiben	806
13.3.2	Parameterlose und parametrisierte Konstruktoren	807
13.3.3	Nützliche vordefinierte Schnittstellen für Konstruktorreferenzen	808
13.4	Funktionale Programmierung	809
13.4.1	Code = Daten	809
13.4.2	Programmierparadigmen: imperativ oder deklarativ	810
13.4.3	Das Wesen der funktionalen Programmierung	811
13.4.4	Funktionale Programmierung und funktionale Programmiersprachen	814
13.4.5	Funktionen höherer Ordnung am Beispiel von Comparator	816
13.4.6	Lambda-Ausdrücke als Abbildungen bzw. Funktionen betrachten	816
13.5	Funktionale Schnittstellen aus dem java.util.function-Paket	818
13.5.1	Blöcke mit Code und die funktionale Schnittstelle Consumer	818
13.5.2	Supplier	820
13.5.3	Prädikate und java.util.function.Predicate	821
13.5.4	Funktionen über die funktionale Schnittstelle java.util.function.Function	823
13.5.5	Ein bisschen Bi	827
13.5.6	Funktionale Schnittstellen mit Primitiven	830
13.6	Optional ist keine Nullnummer	832
13.6.1	Einsatz von null	833
13.6.2	Der Optional-Typ	835
13.6.3	Erst mal funktional mit Optional	838
13.6.4	Primitiv-Optionales mit speziellen Optional*-Klassen	841
13.7	Was ist jetzt so funktional?	843
13.8	Zum Weiterlesen	845

14 Architektur, Design und angewandte Objektorientierung 847

14.1	SOLIDe Modellierung	847
14.1.1	DRY, KISS und YAGNI	848
14.1.2	SOLID	848
14.1.3	Sei nicht STUPID	850
14.2	Architektur, Design und Implementierung	851
14.3	Design-Patterns (Entwurfsmuster)	852
14.3.1	Motivation für Design-Patterns	852

14.3.2	Singleton	853
14.3.3	Fabrikmethoden	854
14.3.4	Das Beobachter-Pattern mit Listener realisieren	855
14.4	Zum Weiterlesen	859

15 Java Platform Module System 861

15.1	Klassenlader (Class Loader) und Modul-/Klassenpfad	861
15.1.1	Klassenladen auf Abruf	861
15.1.2	Klassenlader bei der Arbeit zusehen	862
15.1.3	JMOD-Dateien und JAR-Dateien	863
15.1.4	Woher die Klassen kommen: Suchorte und spezielle Klassenlader	864
15.1.5	Setzen des Suchpfades	865
15.2	Module im JPMS einbinden	867
15.2.1	Wer sieht wen?	868
15.2.2	Plattform-Module und ein JMOD-Beispiel	869
15.2.3	Interne Plattformeigenschaften nutzen, --add-exports	870
15.2.4	Neue Module einbinden	872
15.3	Eigene Module entwickeln	873
15.3.1	Modul com.tutego.candytester	873
15.3.2	Moduldeklaration mit module-info.java und exports	874
15.3.3	Modul com.tutego.main	875
15.3.4	Modulinfodatei mit requires	875
15.3.5	Moduleinsetzer schreiben, JVM-Schalter -p und -m	876
15.3.6	Modulinfodatei-Experimente	877
15.3.7	Automatische Module	878
15.3.8	Unbenanntes Modul	879
15.3.9	Lesbarkeit und Zugreifbarkeit	879
15.3.10	Modul-Migration	880
15.4	Zum Weiterlesen	881

16 Die Klassenbibliothek 883

16.1	Die Java-Klassenphilosophie	883
16.1.1	Modul, Paket, Typ	883
16.1.2	Übersicht über die Pakete der Standardbibliothek	886

16.2 Einfache Zeitmessung und Profiling *	890
16.3 Die Klasse Class	891
16.3.1 An ein Class-Objekt kommen	892
16.3.2 Eine Class ist ein Type	894
16.4 Die Utility-Klassen System und Properties	894
16.4.1 Speicher der JVM	895
16.4.2 Anzahl der CPUs bzw. Kerne	896
16.4.3 Systemeigenschaften der Java-Umgebung	897
16.4.4 Eigene Properties von der Konsole aus setzen *	898
16.4.5 Zeilenumbruchzeichen, line.separator	900
16.4.6 Umgebungsvariablen des Betriebssystems	900
16.5 Sprachen der Länder	902
16.5.1 Sprachen in Regionen über Locale-Objekte	902
16.6 Wichtige Datum-Klassen im Überblick	906
16.6.1 Der 1.1.1970	907
16.6.2 System.currentTimeMillis()	907
16.6.3 Einfache Zeitumrechnungen durch TimeUnit	908
16.7 Date-Time-API	909
16.7.1 Menschenzeit und Maschinenzeit	910
16.7.2 Die Datumsklasse LocalDate	913
16.8 Logging mit Java	914
16.8.1 Logging-APIs	914
16.8.2 Logging mit java.util.logging	914
16.9 Maven: Build-Management und Abhängigkeiten auflösen	917
16.9.1 Dependency hinzunehmen	917
16.9.2 Lokales und das Remote Repository	918
16.9.3 Lebenszyklus, Phasen und Maven-Plugins	918
16.10 Zum Weiterlesen	919

17 Einführung in die nebenläufige Programmierung 921

17.1 Nebenläufigkeit und Parallelität	921
17.1.1 Multitasking, Prozesse und Threads	922
17.1.2 Wie nebenläufige Programme die Geschwindigkeit steigern können	923
17.1.3 Java-Threads	924
17.1.4 Was Java für Nebenläufigkeit alles bietet	925

17.2 Existierende Threads und neue Threads erzeugen	925
17.2.1 Main-Thread	925
17.2.2 Wer bin ich?	926
17.2.3 Die Schnittstelle Runnable implementieren	926
17.2.4 Thread aufbauen	927
17.2.5 Thread mit Runnable starten	928
17.2.6 Runnable parametrisieren	930
17.2.7 Die Klasse Thread erweitern *	931
17.3 Thread-Eigenschaften und Zustände	931
17.3.1 Der Name eines Threads	931
17.3.2 Die Zustände eines Threads *	932
17.3.3 Schläfer gesucht	932
17.3.4 Wann Threads fertig sind	934
17.3.5 Einen Thread höflich mit Interrupt beenden	934
17.3.6 Unbehandelte Ausnahmen, Thread-Ende und UncaughtExceptionHandler	937
17.3.7 Der stop() von außen *	939
17.3.8 Ein Rendezvous mit join(...) *	939
17.3.9 Arbeit niederlegen und wieder aufnehmen *	941
17.3.10 Priorität *	941
17.4 Der Ausführer (Executor) kommt	943
17.4.1 Die Schnittstelle Executor	943
17.4.2 Glücklich in der Gruppe – die Thread-Pools	945
17.4.3 Threads mit Rückgabe über Callable	947
17.4.4 Erinnerungen an die Zukunft – die Future-Rückgabe	949
17.4.5 Mehrere Callable-Objekte abarbeiten	952
17.4.6 CompletionService und ExecutorCompletionService	953
17.4.7 ScheduledExecutorService: wiederholende Aufgaben und Zeitsteuerungen	955
17.4.8 Asynchrones Programmieren mit CompletableFuture (CompletionStage) ...	955
17.5 Zum Weiterlesen	958
 18 Einführung in Datenstrukturen und Algorithmen	 959
18.1 Listen	959
18.1.1 Erstes Listen-Beispiel	960
18.1.2 Auswahlkriterium ArrayList oder LinkedList	961
18.1.3 Die Schnittstelle List	961
18.1.4 ArrayList	968

18.1.5	LinkedList	969
18.1.6	Der Array-Adapter Arrays.asList(...)	971
18.1.7	toArray(...) von Collection verstehen – die Gefahr einer Falle erkennen	973
18.1.8	Primitive Elemente in Datenstrukturen verwalten	976
18.2	Mengen (Sets)	977
18.2.1	Ein erstes Mengen-Beispiel	977
18.2.2	Methoden der Schnittstelle Set	979
18.2.3	HashSet	981
18.2.4	TreeSet – die sortierte Menge	982
18.2.5	Die Schnittstellen NavigableSet und SortedSet	984
18.2.6	LinkedHashSet	986
18.3	Assoziative Speicher	987
18.3.1	Die Klassen HashMap und TreeMap, statische Map-Methoden	988
18.3.2	Einfügen und Abfragen des Assoziativspeichers	991
18.4	Ausgewählte Basistypen	994
18.5	Java-Stream-API	994
18.5.1	Deklaratives Programmieren	994
18.5.2	Interne versus externe Iteration	995
18.5.3	Was ist ein Stream?	996
18.6	Einen Stream erzeugen	997
18.6.1	Parallele oder sequenzielle Streams	1000
18.7	Terminale Operationen	1001
18.7.1	Die Anzahl der Elemente	1001
18.7.2	Und jetzt alle – forEach*(...)	1002
18.7.3	Einzelne Elemente aus dem Strom holen	1002
18.7.4	Existenztests mit Prädikaten	1003
18.7.5	Einen Strom auf sein kleinstes bzw. größtes Element reduzieren	1004
18.7.6	Einen Strom mit eigenen Funktionen reduzieren	1005
18.7.7	Ergebnisse in einen Container schreiben, Teil 1: collect(...)	1006
18.7.8	Ergebnisse in einen Container schreiben, Teil 2: Collector und Collectors	1007
18.7.9	Ergebnisse in einen Container schreiben, Teil 3: Gruppierungen	1009
18.7.10	Stream-Elemente in ein Array oder einen Iterator übertragen	1011
18.8	Intermediäre Operationen	1012
18.8.1	Element-Vorschau	1013
18.8.2	Filtern von Elementen	1013
18.8.3	Statusbehaftete intermediäre Operationen	1014
18.8.4	Präfix-Operation	1015
18.8.5	Abbildungen	1016
18.9	Zum Weiterlesen	1019

19 Einführung in grafische Oberflächen 1021

19.1 GUI-Frameworks	1021
19.1.1 Kommandozeile	1021
19.1.2 Grafische Benutzeroberfläche	1021
19.1.3 Abstract Window Toolkit (AWT)	1022
19.1.4 Java Foundation Classes und Swing	1022
19.1.5 JavaFX	1022
19.1.6 SWT (Standard Widget Toolkit) *	1024
19.2 Deklarative und programmierte Oberflächen	1025
19.2.1 GUI-Beschreibungen in JavaFX	1025
19.2.2 Deklarative GUI-Beschreibungen für Swing?	1026
19.3 GUI-Builder	1026
19.3.1 GUI-Builder für JavaFX	1027
19.3.2 GUI-Builder für Swing	1027
19.4 Mit dem Eclipse WindowBuilder zur ersten Swing-Oberfläche	1027
19.4.1 WindowBuilder installieren	1028
19.4.2 Mit WindowBuilder eine GUI-Klasse hinzufügen	1029
19.4.3 Das Layoutprogramm starten	1031
19.4.4 Grafische Oberfläche aufbauen	1032
19.4.5 Swing-Komponenten-Klassen	1035
19.4.6 Funktionalität geben	1036
19.5 Grundlegendes zum Zeichnen	1039
19.5.1 Die paint(Graphics)-Methode für den AWT-Frame	1040
19.5.2 Die ereignisorientierte Programmierung ändert Fensterinhalte	1041
19.5.3 Zeichnen von Inhalten auf einen JFrame	1043
19.5.4 Auffordern zum Neuzeichnen mit repaint(...)	1044
19.5.5 Java 2D-API	1045
19.6 Zum Weiterlesen	1045

20 Einführung in Dateien und Datenströme 1047

20.1 Alte und neue Welt in java.io und java.nio	1047
20.1.1 java.io-Paket mit File-Klasse	1047
20.1.2 NIO.2 und das java.nio-Paket	1048
20.1.3 java.io.File oder java.nio.*-Typen?	1048

20.2	Dateisysteme und Pfade	1048
20.2.1	FileSystem und Path	1049
20.2.2	Die Utility-Klasse Files	1056
20.3	Dateien mit wahlfreiem Zugriff	1058
20.3.1	Ein RandomAccessFile zum Lesen und Schreiben öffnen	1059
20.3.2	Aus dem RandomAccessFile lesen	1059
20.3.3	Schreiben mit RandomAccessFile	1062
20.3.4	Die Länge des RandomAccessFile	1062
20.3.5	Hin und her in der Datei	1063
20.4	Basisklassen für die Ein-/Ausgabe	1064
20.4.1	Die vier abstrakten Basisklassen	1064
20.4.2	Die abstrakte Basisklasse OutputStream	1065
20.4.3	Die abstrakte Basisklasse InputStream	1067
20.4.4	Die abstrakte Basisklasse Writer	1069
20.4.5	Die Schnittstelle Appendable *	1070
20.4.6	Die abstrakte Basisklasse Reader	1071
20.4.7	Die Schnittstellen Closeable, AutoCloseable und Flushable	1074
20.5	Lesen aus Dateien und Schreiben in Dateien	1076
20.5.1	Byteorientierte Datenströme über Files beziehen	1076
20.5.2	Zeichenorientierte Datenströme über Files beziehen	1077
20.5.3	Die Funktion von OpenOption bei den Files.new*(...)-Methoden	1079
20.5.4	Ressourcen aus dem Modulpfad und aus JAR-Dateien laden	1081
20.6	Zum Weiterlesen	1082

21 Einführung ins Datenbankmanagement mit JDBC 1083

21.1	Relationale Datenbanken und Java-Zugriffe	1083
21.1.1	Das relationale Modell	1083
21.1.2	Java-APIs zum Zugriff auf relationale Datenbanken	1084
21.1.3	Die JDBC-API und Implementierungen: JDBC-Treiber	1085
21.1.4	H2 ist das Werkzeug auf der Insel	1085
21.2	Eine Beispielabfrage	1086
21.2.1	Schritte zur Datenbankabfrage	1086
21.2.2	Mit Java auf die relationale Datenbank zugreifen	1086
21.3	Zum Weiterlesen	1088

22 Bits und Bytes, Mathematisches und Geld	1089
22.1 Bits und Bytes	1089
22.1.1 Die Bit-Operatoren Komplement, Und, Oder und XOR	1090
22.1.2 Repräsentation ganzer Zahlen in Java – das Zweierkomplement	1091
22.1.3 Das binäre (Basis 2), oktale (Basis 8), hexadezimale (Basis 16) Stellenwertsystem	1093
22.1.4 Auswirkung der Typumwandlung auf die Bit-Muster	1094
22.1.5 Vorzeichenlos arbeiten	1097
22.1.6 Die Verschiebeoperatoren	1099
22.1.7 Ein Bit setzen, löschen, umdrehen und testen	1102
22.1.8 Bit-Methoden der Integer- und Long-Klasse	1102
22.2 Fließkomma-Arithmetik in Java	1104
22.2.1 Spezialwerte für Unendlich, Null, NaN	1105
22.2.2 Standardnotation und wissenschaftliche Notation bei Fließkommazahlen *	1108
22.2.3 Mantisse und Exponent *	1108
22.3 Die Eigenschaften der Klasse Math	1110
22.3.1 Objektvariablen der Klasse Math	1110
22.3.2 Absolutwerte und Vorzeichen	1111
22.3.3 Maximum/Minimum	1111
22.3.4 Runden von Werten	1112
22.3.5 Rest der ganzzahligen Division *	1115
22.3.6 Division mit Rundung in Richtung negativ unendlich, alternativer Restwert *	1116
22.3.7 Multiply-Accumulate	1117
22.3.8 Wurzel- und Exponentialmethoden	1118
22.3.9 Der Logarithmus *	1119
22.3.10 Winkelmethode *	1120
22.3.11 Zufallszahlen	1121
22.4 Genauigkeit, Wertebereich eines Typs und Überlaufkontrolle *	1121
22.4.1 Der größte und der kleinste Wert	1122
22.4.2 Überlauf und alles ganz exakt	1122
22.4.3 Was bitte macht eine ulp?	1125
22.5 Zufallszahlen: Random, ThreadLocalRandom und SecureRandom	1127
22.5.1 Die Klasse Random	1127
22.5.2 Die Klasse ThreadLocalRandom	1130
22.5.3 Die Klasse SecureRandom *	1131

22.6 Große Zahlen *	1131
22.6.1 Die Klasse BigInteger	1131
22.6.2 Beispiel: ganz lange Fakultäten mit BigInteger	1138
22.6.3 Große Fließkommazahlen mit BigDecimal	1139
22.6.4 Mit MathContext komfortabel die Rechengenauigkeit setzen	1142
22.6.5 Noch schneller rechnen durch mutable Implementierungen	1144
22.7 Geld und Währung	1144
22.7.1 Geldbeträge repräsentieren	1144
22.7.2 ISO 4217	1145
22.7.3 Währungen in Java repräsentieren	1145
22.8 Zum Weiterlesen	1146

23 Testen mit JUnit 1147

23.1 Softwaretests	1147
23.1.1 Vorgehen beim Schreiben von Testfällen	1148
23.2 Das Test-Framework JUnit	1148
23.2.1 Test-Driven Development und Test-First	1149
23.2.2 Testen, implementieren, testen, implementieren, testen, freuen	1150
23.2.3 JUnit-Tests ausführen	1152
23.2.4 assert*(...)-Methoden der Klasse Assertions	1153
23.2.5 Exceptions testen	1156
23.2.6 Grenzen für Ausführungszeiten festlegen	1157
23.2.7 Beschriftungen mit @DisplayName	1158
23.2.8 Verschachtelte Tests	1158
23.2.9 Tests ignorieren	1159
23.2.10 Mit Methoden der Assumptions-Klasse Tests abbrechen	1159
23.2.11 Parametrisierte Tests	1159
23.3 Java-Assertions-Bibliotheken und AssertJ	1161
23.3.1 AssertJ	1161
23.4 Aufbau größerer Testfälle	1163
23.4.1 Fixtures	1163
23.4.2 Sammlungen von Testklassen und Klassenorganisation	1165
23.5 Wie gutes Design das Testen ermöglicht	1165
23.6 Dummy, Fake, Stub und Mock	1167
23.7 JUnit-Erweiterungen, Testzusätze	1169
23.8 Zum Weiterlesen	1170

24 Die Werkzeuge des JDK	1171
24.1 Übersicht	1171
24.1.1 Aufbau und gemeinsame Schalter	1172
24.2 Java-Quellen übersetzen	1172
24.2.1 Der Java-Compiler des JDK	1172
24.2.2 Native Compiler	1173
24.3 Die Java-Laufzeitumgebung	1173
24.3.1 Schalter der JVM	1173
24.3.2 Der Unterschied zwischen java.exe und javaw.exe	1176
24.4 Dokumentationskommentare mit Javadoc	1176
24.4.1 Einen Dokumentationskommentar setzen	1176
24.4.2 Mit dem Werkzeug javadoc eine Dokumentation erstellen	1179
24.4.3 HTML-Tags in Dokumentationskommentaren *	1179
24.4.4 Generierte Dateien	1179
24.4.5 Dokumentationskommentare im Überblick *	1180
24.4.6 Javadoc und Doclets *	1182
24.4.7 Veraltete (deprecated) Typen und Eigenschaften	1182
24.4.8 Javadoc-Überprüfung mit DocLint	1185
24.5 Das Archivformat JAR	1186
24.5.1 Das Dienstprogramm jar benutzen	1186
24.5.2 Das Manifest	1186
24.5.3 Applikationen in JAR-Archiven starten; ausführbare JAR-Dateien	1187
24.6 Zum Weiterlesen	1188
 Anhang	 1189
A Java SE-Module und Paketübersicht	1189
 Index	 1207

Materialien zum Buch

Auf der Webseite zu diesem Buch stehen folgende Materialien für Sie zum Download bereit:

► **alle Beispielprogramme**

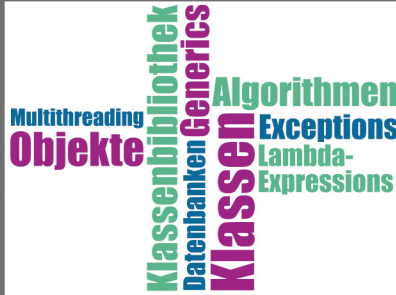
Gehen Sie auf www.rheinwerk-verlag.de/5712. Klicken Sie auf den Reiter MATERIALIEN. Sie sehen die herunterladbaren Dateien samt einer Kurzbeschreibung des Dateiinhalts. Klicken Sie auf den Button HERUNTERLADEN, um den Download zu starten. Je nach Größe der Datei (und Ihrer Internetverbindung) kann es einige Zeit dauern, bis der Download abgeschlossen ist.

Java ist auch eine Insel

»Ideal für
Selbststudium,
Ausbildung und
Beruf«

Java von A bis Z

Das Java-Kultbuch lehrt 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.



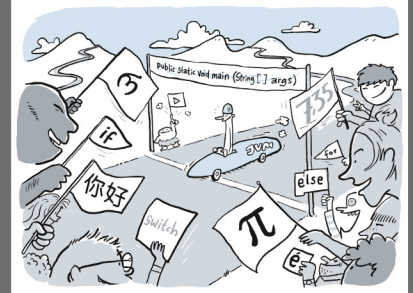
Umfassendes Java-Wissen

Warum übersetzt der Compiler Folgendes ohne Murren?

Listing 2.33 src/main/java/WithoutComplain.java

```
class WithoutComplain {  
    static void main( String[] args ) {  
        http://www.tutego.de/  
        System.out.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. Als Basis dient das freie, quelloffene OpenJDK. Mit Zusatzinfos zu IntelliJ IDEA und Eclipse.

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 21.

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 die anderen 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
- Die Klassenbibliothek

Spezialthemen

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

