Java ist auch eine Insel

Einführung, Ausbildung, Praxis

Kapitel 3
Klassen und Objekte

»Nichts auf der Welt ist so gerecht verteilt wie der Verstand. Denn jedermann ist davon überzeugt, dass er genug davon habe.«
– René Descartes (1596–1650)

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

3.1 Objektorientierte Programmierung (OOP)

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

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

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

Hinweis

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

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

3.1.1 Warum überhaupt OOP?

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

Hinweis

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

Identität, Zustand, Verhalten

Die in der Software abgebildeten Objekte haben drei wichtige Eigenschaften:

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

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

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

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

3.1.2 Denk ich an Java, denk ich an Wiederverwendbarkeit

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

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

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

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

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

3.2 Eigenschaften einer Klasse

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

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

Eine Klasse deklariert im Wesentlichen zwei Dinge:

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

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

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

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

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

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

Hinweis

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

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

3.2.1 Klassenarbeit mit Point

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

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

Tabelle 3.1 OOP-Begriffe und was sie bedeuten

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

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

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

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

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

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

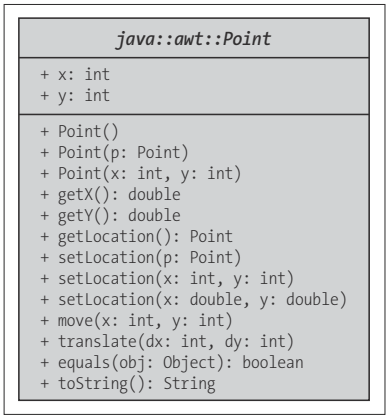

Abbildung 3.1 Die Klasse »java.awt.Point« in der UML-Darstellung

3.3.1 Wichtige Diagrammtypen der UML *

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

Anwendungsfalldiagramm

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

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

Klassendiagramm

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

Objektdiagramm

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

Sequenzdiagramm

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

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

In diesem Buch kommen fast nur Klassendiagramme vor.

3.4 Neue Objekte erzeugen

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

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

3.4.1 Ein Objekt einer Klasse mit dem Schlüsselwort new anlegen

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

Beispiel

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

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

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

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

3.4.2 Deklarieren von Referenzvariablen

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

Beispiel

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

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

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

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

Die Typen müssen natürlich kompatibel sein, und ein `Point`-Objekt geht nicht als `String` durch. Der Versuch, ein 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 Spezialität 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.

zB

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.

eclipse

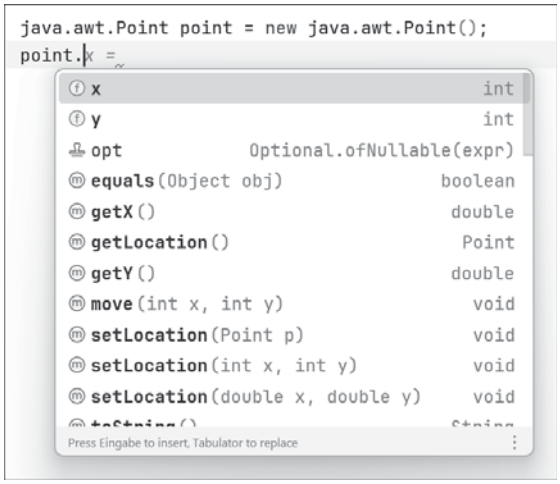

Abbildung 3.3 Die Tastenkombination Strg + Leertaste zeigt an, welche Eigenschaften eine Referenz ermöglicht. Eine Auswahl mit der ↵-Taste wählt die Eigenschaft aus und setzt insbesondere bei Methoden den Cursor zwischen das Klammerpaar.

Tür und Spieler auf dem Spielbrett

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

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

```
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
`int playerX;` `int playerY;`	`java.awt.Point player;`
`int doorX;` `int doorY;`	`java.awt.Point door;`

Tabelle 3.2 Objekte kapseln Zustände.

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

Listing 3.1 src/main/java/PlayerAndDoorAsPoints.java

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

  java.awt.Point door = new java.awt.Point();
  door.setLocation( 10, 100 );

  IO.println( player.distance( door ) );  // 90.0
}
```

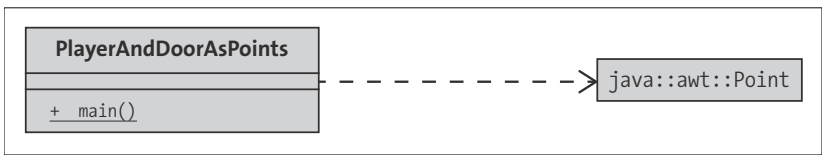

Abbildung 3.4 Die Abhängigkeit zwischen einer Klasse und dem »java.awt.Point« zeigt das UML-Diagramm mit einer gestrichelten Linie an. Attribute und Operationen von »Point« sind nicht dargestellt.

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

toString()

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

Listing 3.2 src/main/java/PointToStringDemo.java

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

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


Tipp

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

Nach dem Punkt geht's weiter

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

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

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

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

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

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

Objekterzeugung ohne Variablenzuweisung

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

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

genau das Gleiche bewirkt wie:

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

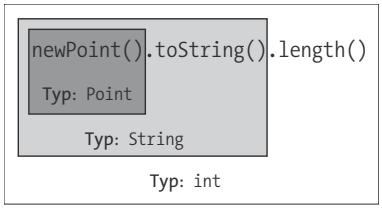

Abbildung 3.5 Jede Schachtelung ergibt einen neuen Typ.

Im Prinzip funktioniert auch Folgendes:

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

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

Beispiel

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

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

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

3.4.4 Überblick über Point-Methoden

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

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

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

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

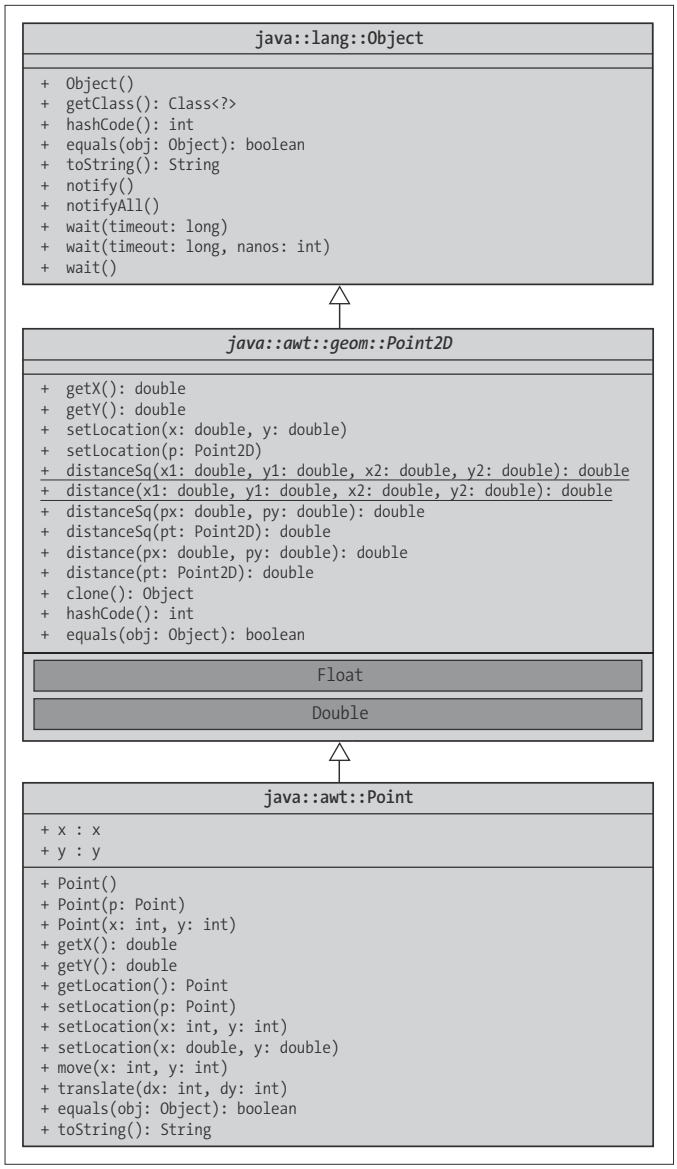

Abbildung 3.6 Vererbungshierarchie bei Point2D

Hinweis

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

Ein paar Worte über Vererbung und die API-Dokumentation *

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

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

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

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

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

Sind zwei Punkte gleich?

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

Listing 3.3 src/main/java/PointEqualsDemo.java

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

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

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

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

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

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

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

Listing 3.4 src/main/java/Distances.java

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

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

```
  IO.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.

3.4.5 Konstruktoren nutzen

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

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

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

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

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

Beispiel

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

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

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

3.5 ZZZZZnake

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

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

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

Im Quellcode sieht das so aus:

Listing 3.5 src/main/java/ZZZZZnake.java

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

  while ( true ) {
    // Draw grid and symbols

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

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

// Determine status

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

3.5.1 Erweiterung

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

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

3.6 Pakete schnüren, Importe und Compilationseinheiten

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

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

3.6.1 Java-Pakete

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

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

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

3.6.2 Pakete der Standardbibliothek

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

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

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

3.6.3 Volle Qualifizierung und import-Deklaration

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

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

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

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

Volle Qualifikation

Listing 3.6 AwtWithoutImport.java

```
static void main(){
 java.awt.Point p =
   new java.awt.Point();

 java.awt.Polygon t =
   new java.awt.Polygon();
 t.addPoint( 10, 10 );
 t.addPoint( 10, 20 );
 t.addPoint( 20, 10 );

 IO.println( p );
 IO.println( t.contains(15,15) );
}
```

import-Deklaration

Listing 3.7 AwtWithImport.java

```
import java.awt.Point;
import java.awt.Polygon;

static void main(){
 Point p = new Point();

 Polygon t = new Polygon();

 t.addPoint( 10, 10 );
 t.addPoint( 10, 20 );
 t.addPoint( 20, 10 );

 IO.println( p );
 IO.println( t.contains(15,15) );
}
```

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

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

Hinweis

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

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

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

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

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

Hinweis

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

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

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

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

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

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

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

Best Practice

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

3.6.5 Modul-Import

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

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

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

Erklärung

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

Die Syntax eines Modul-Imports lautet wie folgt:

```
import module Modulname;
```

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

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

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

Namenskonflikte durch Mehrfach-Importe

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

Nehmen wir folgenden Modul-Import an:

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

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

Es gibt unterschiedliche Lösungsansätze für das Problem.

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

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

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

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

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

Lösung 3: Vollständige Qualifikation von Typen:

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

Und dann zum Beispiel:

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

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

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

Strukturierung

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

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

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

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

class Application { }
```

Automatischer Modul-Import in kompakten Java-Quellcodedateien

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

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

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

3.6.6 Hierarchische Strukturen über Pakete und die Spiegelung im Dateisystem

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

Beispiel

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

Der Aufbau von Paketnamen

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

3.6.7 Die package-Deklaration

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

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

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

Hier der vollständige Code für die Klasse `DatePrinter`:

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

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

import java.time.LocalDate;
```

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

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

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

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

Hinweis

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

Klasse verwenden

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

Die erste Variante sieht so aus:

Listing 3.9 src/main/java/DatePrinterUser1.java

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

Und hier ist die Variante mit der `import`-Deklaration:

Listing 3.10 src/main/java/DatePrinterUser2.java

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

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

Tipp

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

3.6.8 Unbenanntes Paket (default package)

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

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

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

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

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

3.6.9 Compilationseinheit (Compilation Unit)

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

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

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

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

3.6.10 Statischer Import *

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

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

Beispiel

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

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

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

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

Listing 3.11 src/main/java/com/tutego/insel/oop/StaticImport.java

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

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

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

Mehrere Typen statisch importieren

Der statische Import

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

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

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

Best Practice

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

3.7 Mit Referenzen arbeiten, Vielfalt, Identität, Gleichwertigkeit

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

3.7.1 null-Referenz und die Frage der Philosophie

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

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

Beispiel

Deklaration und Initialisierung zweier Objektvariablen mit `null`:

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

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

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

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

Hinweis

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

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

Auf null geht nix, nur die NullPointerException

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

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

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

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

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

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

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

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

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

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

3.7.2 Alles auf null? Referenzen testen

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

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

null-Test und Kurzschluss-Operatoren

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

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

Listing 3.13 src/main/java/NullCheckForStringLength.java

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

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

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

Das Glück der anderen: Null Coalescing Operator *

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

3.7.3 Zuweisungen bei Referenzen

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

Beispiel

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

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

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

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

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

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

```
import java.awt.*;

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

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

3.7.4 Methoden mit Referenztypen als Parameter

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

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

Listing 3.15 src/main/java/DrawPlayerAndSnake.java

```
import java.awt.Point;

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

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

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

Die Ausgabe kann so aussehen:

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

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

Hinweis

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

Wertübergabe und Referenzübergabe per Call by Value

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

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

```
import java.awt.Point;

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

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

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

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


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

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

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

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

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

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

Final deklarierte Referenzparameter und das fehlende const

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

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

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

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

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

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

Sprachenvergleich *

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

3.7.5 Identität von Objekten

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

Beispiel

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

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

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

3.7.6 Gleichwertigkeit und die Methode equals(...)

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

Beispiel

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

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

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

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

	Getestet mit	Implementierung
Identität der Referenzen	`==` bzw. `!=`	nichts zu tun
Gleichwertigkeit der Zustände	`equals(…)` bzw. `! equals(…)`	abhängig von der Klasse

Tabelle 3.4 Identität und Gleichwertigkeit von Objekten

equals(…)-Implementierung von Point *

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

Listing 3.17 java/awt/Point.java

```
public class Point … {

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

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

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

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

Listing 3.18 java/lang/Object.java

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

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

Sprachvergleich

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

3.7.7 Ausblick: Value Types – Objekte ohne Identität

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

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

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

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

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

3.8 Der Zusammenhang von new, Heap und Garbage-Collector

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

3.8.1 Heap-Speicher

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

Hinweis

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

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

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

Der Heap wächst von einer Startgröße bis zu einer maximal erlaubten Größe, um sicherzustellen, dass ein Java-Programm nicht unbeschränkt viel Speicher vom Betriebssystem abruft, was die Maschine möglicherweise in den Ruin treibt. In der HotSpot-JVM beträgt die Startgröße des Heaps $^1/_{64}$ des Hauptspeichers und wächst dann bis zu einer maximalen Größe von $^1/_4$ des Hauptspeichers.[12]

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

Nehmen wir folgendes Szenario an:

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

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

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

3.8.3 OutOfMemoryError

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

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

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

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

Beispiel

Wir provozieren einen `OutOfMemoryError` mit folgender Zeile:

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

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

3.9 Zum Weiterlesen

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

Kapitel 16
Die Klassenbibliothek

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

16.1 Die Java-Klassenphilosophie

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

16.1.1 Modul, Paket, Typ

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

Module der Java SE

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

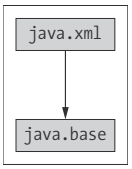

Abbildung 16.1 Das Modul »java.xml« hat eine Abhängigkeit zum »java.base«-Modul.

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

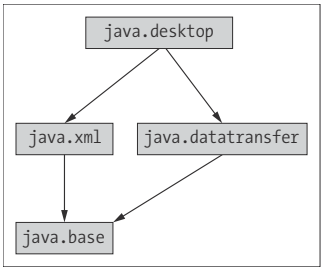

Abbildung 16.2 Abhängigkeiten des Moduls »java.desktop«

Das Modul java.se

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

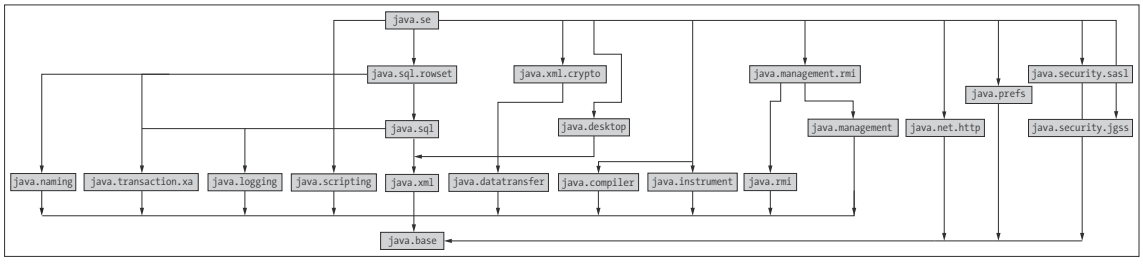

Abbildung 16.3 Abhängigkeiten des Moduls »java.se«

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

Hinweis

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

16.1.2 Übersicht über die Pakete der Standardbibliothek

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

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

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

Tabelle 16.1 Wichtige Pakete in der Java SE

Paket	Beschreibung
`javax.swing`	Swing-Komponenten für grafische Oberflächen. Das Paket besitzt diverse Unterpakete.

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

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

Offizielle Schnittstelle (java- und javax-Pakete)

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

- Das Oracle-Entwicklerteam setzt neue Pakete und Typen in die API.
- Der *Java Community Process* (JCP) beschließt eine neue API. Dann ist es nicht nur Oracle allein, sondern eine Gruppe, die eine neue API erarbeitet und die Schnittstellen definiert.
- Das *World Wide Web Consortium* (W3C) gibt eine API etwa für XML-DOM vor.

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

Standard Extension API (javax-Pakete)

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

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

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

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

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

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

16.2 Einfache Zeitmessung und Profiling *

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

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

- `static long currentTimeMillis()`
  Gibt die seit dem 1.1.1970, 00:00:00 UTC vergangenen Millisekunden zurück.
- `static long nanoTime()`
  Liefert die Zeit vom genauesten Systemzeitgeber. Sie hat keinen Bezugspunkt zu einem Datum.

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

Tipp

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

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

16.2.1 Profiler *

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

16.3 Die Klasse Class

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

In diesem Punkt unterscheidet sich Java von vielen herkömmlichen Programmiersprachen, da sich Eigenschaften von Klassen vom gerade laufenden Programm mithilfe der `Class`-Objekte abfragen lassen. Bei den Exemplaren von `Class` handelt es sich um eine eingeschränkte Form von Meta-Objekten[2] – die Beschreibung eines Java-Typs, die aber nur ausgewählte Informationen preisgibt. Neben normalen Klassen werden auch Schnittstellen durch ein `Class`-Objekt repräsentiert und sogar Arrays und primitive Datentypen – statt `Class` wäre wohl der Klassenname `Type` passender gewesen.

16.3.1 An ein Class-Objekt kommen

Zunächst müssen wir für eine bestimmte Klasse das zugehörige `Class`-Objekt erfragen. `Class`-Objekte selbst kann nur die JVM erzeugen. Wir können das nicht (die Objekte sind 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.

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

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

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

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

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

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

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

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

Beispiel

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

`class java.lang.Object`

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

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

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

ClassNotFoundException und NoClassDefFoundError *

Eine `ClassNotFoundException` lösen die Methoden

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

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

Neben der `ClassNotFoundException` gibt es einen `NoClassDefFoundError` – ein harter `LinkageError`, 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 geglückten Ladeversuch den `NoClassDefFoundError` aus. Auch tritt der Fehler auf, wenn beim Laden des Bytecodes die Klasse `MeineKlasse` zwar gefunden wurde, aber `MeineKlasse` einen statischen Initialisierungsblock besitzt, der wiederum eine Klasse referenziert, für die keine Klassendatei vorhanden ist.

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

Probleme nach Anwendung eines Obfuscators *

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

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

16.3.2 Eine Class ist ein Type

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

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

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

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

16.4 Die Utility-Klassen System und Properties

In der Klasse `java.lang.System` finden sich Methoden zum Erfragen und Ändern von Systemvariablen, zum Umlenken der Standarddatenströme, zum Ermitteln der aktuellen Zeit, zum Beenden der Applikation und noch für das eine oder andere. Alle Methoden sind ausschließlich statisch, und ein Exemplar von `System` lässt sich nicht anlegen. In der Klasse `java.lang.Runtime` finden sich zusätzlich Hilfsmethoden, wie für das Starten von externen Programmen oder Methoden zum Erfragen des Speicherbedarfs. Anders als `System` ist hier nur eine Methode statisch, nämlich die Singleton-Methode `getRuntime()`, die das Exemplar von `Runtime` liefert.

Bemerkung

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

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

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

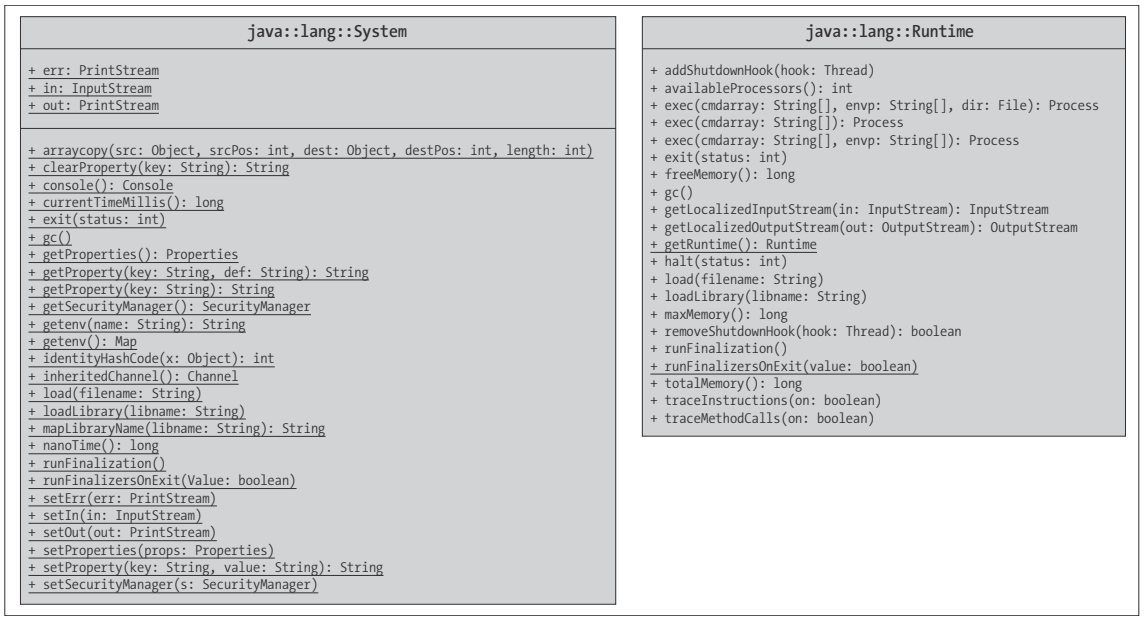

Abbildung 16.4 Eigenschaften der Klassen »System« und »Runtime«

16.4.1 Speicher der JVM

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

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

Zwei nützliche Größen können berechnet werden.

Benutzter Speicher:

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

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

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

Beispiel

Gib Informationen über den Speicher auf einem Rechner aus:

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

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

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

Die Ausgabe kann sein:

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

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

16.4.2 Systemeigenschaften der Java-Umgebung

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

Beispiel

Gib den Namen des Betriebssystems aus:

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

Gib alle Systemeigenschaften auf dem Bildschirm aus:

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

Die Ausgabe beginnt mit:

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

Tabelle 16.2 zeigt eine Liste der wichtigsten Standardsystemeigenschaften:

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

Tabelle 16.2 Standardsystemeigenschaften

API-Dokumentation

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

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

- `static String getProperty(String key)`
  Gibt die Belegung einer Systemeigenschaft zurück. Ist der Schlüssel `null` oder leer, gibt es eine `NullPointerException` bzw. eine `IllegalArgumentException`.

- `static String getProperty(String key, String def)`
  Gibt die Belegung einer Systemeigenschaft zurück. Ist sie nicht vorhanden, liefert die Methode die Zeichenkette `def`, den Default-Wert. Für die Ausnahmen gilt das Gleiche wie bei `getProperty(String)`.
- `static String setProperty(String key, String value)`
  Belegt eine Systemeigenschaft neu. Die Rückgabe ist die alte Belegung – oder `null`, falls es keine alte Belegung gab.
- `static String clearProperty(String key)`
  Löscht eine Systemeigenschaft aus der Liste. Die Rückgabe ist die alte Belegung – oder `null`, falls es keine alte Belegung gab.
- `static Properties getProperties()`
  Liefert ein mit den aktuellen Systembelegungen gefülltes `Properties`-Objekt.

16.4.3 Eigene Properties von der Konsole aus setzen *

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

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

Die Property `LOG` ist einfach nur »da«, aber ohne zugewiesenen Wert. Die nächsten beiden Properties, `USER` und `SIZE`, sind mit Werten verbunden, die erst einmal vom Typ `String` sind und vom Programm weiterverarbeitet werden müssen. Die Informationen tauchen nicht bei der Argumentliste in der statischen `main(…)`-Methode auf, da sie vor dem Namen der Klasse stehen und bereits von der Java-Laufzeitumgebung verarbeitet werden.

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

Listing 16.3 com/tutego/insel/lang/SetProperty.java

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

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

Wir bekommen über `getProperty(String)` einen String zurück, der den Wert anzeigt. Falls es überhaupt keine Eigenschaft dieses Namens gibt, erhalten wir stattdessen `null`. So wissen wir auch, ob dieser Wert überhaupt gesetzt wurde. Ein einfacher `null`-Test sagt also aus, ob `log-`

Property vorhanden ist oder nicht. Statt -DLOG führt auch -DLOG= zum gleichen Ergebnis, denn der assoziierte Wert ist der Leer-String. Da alle Properties erst einmal vom Typ String sind, lässt sich usernameProperty einfach ausgeben, und wir bekommen entweder null oder den hinter = angegebenen String. Sind die Typen keine Strings, müssen sie weiterverarbeitet werden, also etwa mit Integer.parseInt(), Double.parseDouble() usw. Nützlich ist die Methode System.getProperty(String, String), der zwei Argumente übergeben werden, denn das zweite Argument steht für einen Default-Wert. So kann immer ein Standardwert angenommen werden.

Boolean.getBoolean(String)

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

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

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

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

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

Gegenüber einer eigenen, direkten System-Anfrage hat getBoolean(String) auch den Nachteil, dass wir bei der Rückgabe false nicht unterscheiden können, ob es die Eigenschaft schlichtweg nicht gibt oder ob die Eigenschaft mit dem Wert false belegt ist. Auch falsch gesetzte Werte wie -DP=fa1se ergeben immer false.[3]

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

- static boolean getBoolean(String name)
  Liest eine Systemeigenschaft mit dem Namen name aus und liefert true, wenn der Wert der Property gleich dem String "true" ist. Der Vergleich mit "true" erfolgt ohne Beachtung der Groß-/Kleinschreibung. Gibt false zurück, wenn der Wert nicht "true" ist, nicht existiert oder null ist.

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

16.4.4 Zeilenumbruchzeichen, line.separator

Um nach dem Ende einer Zeile an den Anfang der nächsten zu gelangen, wird ein *Zeilenumbruch* (engl. *new line*) eingefügt. Das Zeichen für den Zeilenumbruch muss kein einzelnes sein, es können auch mehrere Zeichen nötig sein. Zum Leidwesen vieler ist die Darstellung des Zeilenumbruchs auf den gängigen Architekturen unterschiedlich:

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

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

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

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

16.4.5 Umgebungsvariablen des Betriebssystems

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

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

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

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

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

- `static String getenv(String name)`
  Liest eine Umgebungsvariable mit dem Namen `name`. Gibt es sie nicht, ist die Rückgabe `null`.

Name der Variablen	Beschreibung	Beispiel
`COMPUTERNAME`	Name des Computers	*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 16.3 Auswahl einiger unter Windows verfügbarer Umgebungsvariablen

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

Beispiel

Lies die zugänglichen Umgebungsvariablen aus und gib sie aus:

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

16.5 Sprachen der Länder

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

16.5.1 Sprachen in Regionen über Locale-Objekte

In Java repräsentieren `Locale`-Objekte Sprachen in geografischen, politischen oder kulturellen Regionen. Die Sprache und die Region müssen getrennt werden, denn nicht immer gibt eine Region oder ein Land die Sprache eindeutig vor. Für Kanada in der Umgebung von Quebec ist die französische Ausgabe relevant, und die unterscheidet sich von der englischen. Jede dieser sprachspezifischen Eigenschaften ist in einem speziellen Objekt gekapselt. `Locale`-Objekte werden dann zum Beispiel einem `Formatter`, der hinter `String.format(…)` und `printf(…)` steht, oder einem `Scanner` übergeben. Diese Ausgaben nennen sich auf Englisch *locale-sensitive*.

Locale-Objekte aufbauen

`Locale`-Objekte werden immer mit dem Namen der Sprache und optional mit dem Namen des Landes bzw. einer Region und Variante erzeugt. Die `Locale`-Klasse bietet mehrere Möglichkeiten[4] zum Aufbau der Objekte:

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

Beispiel

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

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

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

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

Die Sprachen werden durch Zwei-Buchstaben-Kürzel aus dem ISO-639-Code[5] (*ISO Language Code*) identifiziert, und die Ländernamen sind Zwei-Buchstaben-Kürzel, die in ISO 3166[6] (*ISO Country Code*) beschrieben sind.

Beispiel

Drei Varianten zum Aufbau der `Locale.JAPANESE`:

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

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

Konstanten für einige Sprachen

Die `Locale`-Klasse besitzt Konstanten für häufig vorkommende Sprachen, optional mit Ländern. Unter den Konstanten für Länder und Sprachen sind: `CANADA`, `CANADA_FRENCH`, `CHINA` ist gleich `CHINESE` (und auch `PRC` bzw. `SIMPLIFIED_CHINESE`), `ENGLISH`, `FRANCE`, `FRENCH`, `GERMAN`, `GERMANY`, `ITALIAN`, `ITALY`, `JAPAN`, `JAPANESE`, `KOREA`, `KOREAN`, `TAIWAN` (ist gleich `TRADITIONAL_CHINESE`), `UK` und `US`. Manche Konstanten wirken dabei doppelt oder synonym. Die scheinbaren Dopplungen entstehen dadurch, dass `Locale` nicht nur verschiedene Sprachen unterscheidet, sondern diese zusätzlich mit Regionen oder Schriftsystemvarianten kombiniert.

Methoden, die Locale-Exemplare annehmen

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

5 *https://de.wikipedia.org/wiki/Liste_der_ISO-639-1-Codes*
6 *https://de.wikipedia.org/wiki/ISO-3166-1-Kodierliste*
7 Auf Englisch beschreibt das Java-Tutorial von Oracle die Erweiterungen unter *http://docs.oracle.com/javase/tutorial/i18n/locale/index.html*.

Tipp

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

Methoden von Locale *

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

Beispiel

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

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

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

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

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

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

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

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

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

- `String getCountry()`
  Liefert das Länderkürzel nach dem ISO-3166-zwei-Buchstaben-Code.
- `String getLanguage()`
  Liefert das Kürzel der Sprache im ISO-639-Code.
- `String getISO3Country()`
  Liefert die ISO-Abkürzung des Landes dieser Einstellungen und löst eine `MissingResourceException` aus, wenn die ISO-Abkürzung nicht verfügbar ist.
- `String getISO3Language()`
  Liefert die ISO-Abkürzung der Sprache dieser Einstellungen und löst eine `MissingResourceException` aus, wenn die ISO-Abkürzung nicht verfügbar ist.
- `String getVariant()`
  Liefert das Kürzel der Variante oder einen leeren String.

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

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

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

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

16.6 Wichtige Datum-Klassen im Überblick

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

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

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

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

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

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

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

16.6.1 Der 1.1.1970

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

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

16.6.2 System.currentTimeMillis()

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

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

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

16.6.3 Einfache Zeitumrechnungen durch TimeUnit

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

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

Beispiel

Konvertiere 23.746.387 Millisekunden in Stunden:

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

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

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

- `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, `SECONDS`, `MINUTES`, `HOURS`, `DAYS`
  Aufzählungskonstanten von `TimeUnit`
- `long toDays(long duration)`
- `long toHours(long duration)`
- `long toMicros(long duration)`
- `long toMillis(long duration)`
- `long toMinutes(long duration)`
- `long toNanos(long duration)`
- `long toSeconds(long duration)`
- `long convert(long sourceDuration, TimeUnit sourceUnit)`
  Liefert `sourceUnit.to*(sourceDuration)`, wobei `*` für die jeweilige Einheit steht. Beispielsweise liefert es **`HOURS`**`.convert(sourceDuration, sourceUnit)`, dann `sourceUnit.to`**`Hours`**`(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.

16.7.1 Erster Überblick

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

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

Tabelle 16.4 Alle temporalen Klassen aus »java.time«

16.7.2 Menschenzeit und Maschinenzeit

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

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

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

Beispiel

```
LocalDate now = LocalDate.now();
IO.println( now ); // z. B. 2026-01-31
System.out.printf( "%d. %s %d%n",
                   now.getDayOfMonth(), now.getMonth(), now.getYear() );
// z. B. 31. JANUARY 2026
LocalDate bdayMLKing = LocalDate.of( 1929, Month.JANUARY, 15 );
DateTimeFormatter formatter =
 DateTimeFormatter.ofLocalizedDate( FormatStyle.MEDIUM );
IO.println( bdayMLKing.format( formatter ) ); // 15. Januar 1929
```

Die Methode `getMonth()` auf einem `LocalDate` liefert als Ergebnis ein `java.time.Month`-Objekt, und das sind Aufzählungen. Die `toString()`-Repräsentation liefert die Konstante in Großbuchstaben.

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

Beispiel

Ausgabe beim japanischen Kalender:

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

Paketübersicht

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

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

Designprinzipien

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

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

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

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

Tabelle 16.5 Namensmuster in der Date-Time-API

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

16.7.3 Die Datumsklasse LocalDate

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

Ein temporales Objekt kann über die statischen `of(…)`-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. Des Weiteren gibt es `long toEpochDay()` und `long toEpochSecond(LocalTime time, ZoneOffset offset)`.

Beispiel

Finde von heute aus gesehen den nächsten Samstag:

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

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

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

16.8 Logging mit Java

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

16.8.1 Logging-APIs

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

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

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

16.8.2 Logging mit java.util.logging

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

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

```
package com.tutego.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 LoggerDemo {

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

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

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

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

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

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

Juli 13, 2025 1:40:15 PM com.tutego.insel.logging.LoggerDemo main
INFORMATION: Runtime: 229 ms
```

Das Logger-Objekt

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

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

Loggen mit Log-Level

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

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

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

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

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

Tabelle 16.6 Log-Level und Methoden

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

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

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

16.9 Maven: Build-Management und Abhängigkeiten auflösen

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

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

16.9.1 Dependency hinzunehmen

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

Listing 16.6 pom.xml

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

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

Alles ist vorbereitet, Zeit für das Hauptprogramm:

Listing 16.7 src/main/java/SparkServer.java

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

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

16.9.2 Lokales und das Remote Repository

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

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

IJ

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

16.9.3 Lebenszyklus, Phasen und Maven-Plugins

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

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

```
$ mvn package
```

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

16.10 Zum Weiterlesen

Die Java-Bibliothek bietet zwar reichlich Klassen und Methoden, aber nicht immer das, was das aktuelle Projekt gerade benötigt. Die Lösung von Problemen, wie etwa dem Aufbau und der Konfiguration von Java-Projekten, objektrelationalen Mappern (*www.hibernate.org*) oder Kommandozeilenparsern, liegt in diversen kommerziellen oder quelloffenen Bibliotheken und Frameworks. Während bei eingekauften Produkten die Lizenzfrage offensichtlich ist, ist bei quelloffenen Produkten eine Integration in das eigene Closed-Source-Projekt nicht immer selbstverständlich. Diverse Lizenzformen (*https://opensource.org/licenses*) bei Open-

Source-Software mit immer unterschiedlichen Vorgaben – Quellcode veränderbar, Derivate müssen frei sein, Vermischung mit proprietärer Software möglich – erschweren die Auswahl, und Verstöße (*https://gpl-violations.org/*) werden öffentlich angeprangert und sind unangenehm. Wer in Java entwickelt, sollte für den kommerziellen Vertrieb sein 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 eine Milliarde Quellcodezeilen verschiedener Programmiersprachen ermöglicht; erstaunlich, wie viele »F*ck« schreiben. Und »Porn Groove« kannte ich vor dieser Suche auch noch nicht.