Kapitel 5
Klassen und Objekte

Java ist eine objektorientierte Sprache, Objekte sind das Herzstück der Sprache. In diesem Kapitel kommen Sie endlich dazu, eigene Objekte anzulegen und mit ihnen zu arbeiten. Außerdem werden Sie endlich mehr zu Themen erfahren, die Sie in den Beispielen der vorigen Kapitel noch als gegeben hinnehmen mussten: Methoden, Access Modifier und mehr.

Objekte sind, wie unschwer am Wort zu erkennen ist, die Grundlage einer objektorientierten Sprache wie Java. Sie sind Datenstrukturen, die mehr können, als nur Daten speichern. Natürlich tun sie auch das, die gespeicherten Daten heißen der *Zustand* des Objekts. Aber sie bieten außerdem *Operationen* an, Möglichkeiten, auf den Zustand zuzugreifen und ihn zu verändern. Das geschieht über eine wohldefinierte Schnittstelle.

Ein Beispiel für eine Datenstruktur, die Sie vielleicht schon kennen, ist die Liste: Sie speichert beliebig viele Datensätze in der Reihenfolge, in der sie der Liste hinzugefügt wurden. Ohne auf Details einzugehen, wie eine solche Liste implementiert wird, kann eine Liste als Datenstruktur auch in nicht objektorientierten Sprachen implementiert werden. Diese Liste enthält dann nur ihre Daten, aber die Operationen – ein Element hinzufügen, entfernen, prüfen, ob ein Element enthalten ist – werden außerhalb der Datenstruktur definiert, als Funktionen, denen eine Liste als Parameter übergeben wird. Das funktioniert, hat aber Nachteile. So können Sie an der Datenstruktur nicht sehen, welche Operationen sie kennt, sondern müssen passende Funktionen selbst suchen. Und wenn es mehrere Arten von Listen gibt, beispielsweise *Linked Lists* (verkettete Listen) und *Array Lists* (dynamische Arrays), dann können Sie diese nicht einfach gleich behandeln, sondern müssen selbst entscheiden, um welche Art von Liste es sich handelt, und müssen die passende Funktion dazu finden.

Im Gegensatz dazu führt eine Liste als Objekt die Operationen, die sie unterstützt, selbst mit. So fällt es leicht, herauszufinden, welche Operationen das sind, und dank der Polymorphie-Eigenschaft (siehe nächstes Kapitel) müssen Sie auch keinen Unterschied mehr machen, welche Art von Liste Sie gerade betrachten.

Dadurch, dass der Zugriff auf den Zustand nur durch die definierte Schnittstelle möglich ist, können die gespeicherten Daten auch nicht mehr in unerlaubter Weise ver-

ändert werden. In einer reinen Datenstruktur, auf deren Inhalt voller Zugriff von außen möglich ist, kann ein fehlerhafter Zugriff dazu führen, dass die Struktur beschädigt wird. Bei Objekten ist dies so gut wie ausgeschlossen, wenn sie korrekt definiert sind.

5.1 Klassen und Objekte

Es gibt zwei grundlegende Konzepte für die Objektorientierung in Java: *Klassen* und *Objekte*. Beim Thema Objekttypen haben Sie bereits kurz gelesen, was ein Objekt überhaupt ist: eine Datenstruktur, die zusammengehörige Daten und Operationen zusammenfasst.

Eine Klasse ist die Definition eines Objekttyps. Die Klasse beschreibt, welche Daten und Operationen ein Objekt hat. Sie ist eine Blaupause für alle Objekte dieses Typs. Die Klasse ist das, was Sie programmieren. Objekte werden mit dem `new`-Operator auf Basis einer Klasse erzeugt.

Im Gegensatz dazu kann es beliebig viele Objekte eines Typs geben. Man nennt ein Objekt, das mit `new` aus einer Klasse erzeugt wurde, eine *Instanz* dieser Klasse.

Die Daten eines Objekts werden in *Feldern* gespeichert. So heißen Variablen, die zu einem Objekt gehören und nicht, wie die bisher verwendeten lokalen Variablen, zu einer Methode. Die Operationen eines Objekts werden durch die *Methoden* des Objekts zur Verfügung gestellt. Methoden arbeiten mit den Daten des Objekts, sie können die Daten verändern und sie dem Aufrufer der Methode zugänglich machen. Felder und Methoden heißen auch die *Member* einer Klasse.

Die Felder eines Objekts sind unabhängig von den Feldern aller anderen Objekte. Änderungen, die Sie an den Daten eines Objekts vornehmen, haben keine Auswirkung auf andere Objekte.

5.1.1 Klassen anlegen

Eine Klasse anzulegen ist nichts Neues mehr für Sie, bei der Lösung jeder Übung haben Sie bereits genau das getan. Eine Klassendeklaration besteht einfach aus dem Schlüsselwort `class`, gefolgt vom Klassennamen, gefolgt vom Rumpf der Klasse in geschweiften Klammen:

```
public class Song {
    //hier stehen Ihre Felder und Methoden
}
```

Listing 5.1 Klassendeklaration

In diesem Beispiel und auch in allen bisherigen Übungen geht der Klassendeklaration der Access Modifier `public` voran: Diese Klasse kann in jeder beliebigen anderen Klasse verwendet werden, zum Beispiel können dort Felder vom Typ dieser Klasse deklariert werden.

Das trifft für die Mehrheit der Klassendeklarationen zu, muss aber nicht so sein, Sie können auch Klassen mit anderen Sichtbarkeiten deklarieren. Von dieser Möglichkeit werden Sie vorwiegend bei *inneren Klassen* Gebrauch machen, Klassen, die innerhalb von anderen Klassen deklariert werden, zu denen Sie mehr in Abschnitt 6.4, »Innere Klassen«, finden.

Klassen mit dem Access Modifier `public` müssen in einer Datei definiert werden, deren Namen dem Klassennamen entspricht (Ausnahmen sind die eben erwähnten inneren Klassen). Da Klassennamen innerhalb eines `package` eindeutig sein müssen, folgt daraus, dass eine Datei nur eine öffentliche (`public`) Klasse enthalten kann.

5.1.2 Objekte erzeugen

Ein Objekt zu instanziieren ist eine Kleinigkeit, Sie brauchen dazu bloß den `new`-Operator:

```
Song aSong = new Song();
```

Listing 5.2 Objektinstanziierung

Beachten Sie, dass dem Klassennamen Klammern folgen. Runde Klammern deuten auf einen Methodenaufruf hin, und genau darum handelt es sich hier auch: Mit dem `new`-Operator rufen Sie einen *Konstruktor* auf, eine spezielle Art von Methode, deren Aufgabe es ist, das neue Objekt zu initialisieren. Wird ein neues Objekt erzeugt, geschieht dies *immer* durch einen Konstruktor, selbst wenn Sie in Ihrer Klasse keinen deklarieren.

Namenskonventionen

Es gibt in Java Namenskonventionen, die zwar nicht vom Compiler erzwungen werden, die Sie aber dennoch einhalten sollten, um Ihren Code möglichst leicht verständlich zu halten:

- **Klassen** werden immer mit einem führenden Großbuchstaben benannt. Besteht ein Klassenname aus mehreren Wörtern, werden diese zusammengeschrieben, und der erste Buchstabe jedes Wortes wird großgeschrieben. Leerzeichen werden nicht durch Unterstriche oder sonstige Zeichen ersetzt. Diese Schreibweise wird als *CamelCase* bezeichnet. Beispiele: `Song`, `KaraokeSong`.
- **Feldnamen** beginnen immer mit einem Kleinbuchstaben, ansonsten wird auch für sie CamelCase angewendet. Beispiel: `titel`, `laengeInSekunden`.

- **Methoden** werden nach denselben Konventionen benannt wie Felder. Ein Methodenname sollte außerdem im Imperativ stehen, zum Beispiel `ladeDaten`, `starte`, `tuEtwas` oder auf Englisch `loadData`, `start`, `doStuff`.
- **Konstanten** (siehe Abschnitt 5.8.2, »Konstanten«) werden nur in Großbuchstaben benannt, Leerzeichen werden durch Unterstriche ersetzt. Beispiel: `UNBEKANNTER_KUENSTLER`, `MAX_VALUE`.

5.2 Access Modifier

Bei der Arbeit mit Objekten werden die verschiedenen Access Modifier wichtig, denn sie steuern, von wo auf Felder und Methoden eines Objekts zugegriffen werden kann (siehe Tabelle 5.1).

Modifier	Wer darf zugreifen?
`public`	Auf Member mit dem `public`-Modifier darf ohne Einschränkungen von überall aus zugegriffen werden.
`protected`	Zugriff ist möglich für Klassen, die im gleichen Package liegen, sowie für Klassen, die von dieser Klasse erben. Mehr zu Vererbung finden Sie in Kapitel 6, »Objektorientierung«.
- (Package-Zugriff)	Wenn Sie keinen Access Modifier angeben, dann ist das Feld oder die Methode für alle Klassen sichtbar, die im gleichen Package liegen. Das ist aber kein Schutz gegen Zugriffe von außerhalb auf Ihre Klasse: Packages in Java sind offen, jeder kann eine weitere Klasse in Ihrem Package deklarieren und von dort auf diese Felder zugreifen.
`private`	Nur die Klasse selbst darf auf `private`-Felder zugreifen. Wie alle Access Modifier wirkt auch `private` auf Klassenebene. Das bedeutet, dass ein Objekt auf `private`-Member eines anderen Objekts des gleichen Typs zugreifen kann.

Tabelle 5.1 Die vier Access Modifier

Im Zusammenhang mit Access Modifiern spricht man oft auch von der Sichtbarkeit von Feldern und Methoden. Das bedeutet bei Feldern aber nicht, dass auf sie nur lesend zugegriffen werden kann. Dass ein Feld sichtbar ist bedeutet in diesem Zusammenhang immer, dass es gelesen und beschrieben werden kann. Es gibt über Access Modifier keine Möglichkeit, die *Art* des Zugriffs auf ein Feld einzuschränken.

Wenn Sie den Zugriff derart einschränken wollen, dann verwenden Sie dazu *Zugriffsmethoden* (siehe Abschnitt 5.4.5).

Die Frage, welche Sichtbarkeit ein Member haben sollte, ist nicht immer einfach zu beantworten, aber es gibt einige solide Faustregeln. Alles, was zur *öffentlichen API* (Application Programming Interface) Ihrer Klasse gehört, sollte den Modifier `public` tragen.

Was ist eine API?

Die API einer Klasse ist die am Anfang des Kapitels angesprochene wohldefinierte Schnittstelle, die Gesamtheit aller Operationen, die die Klasse anbietet. Dazu gehören alle Methoden, die ein Benutzer der Klasse verwenden kann, um auf ihren Zustand zuzugreifen (»Benutzer« bedeutet in diesem Zusammenhang eine andere Klasse, die diese Klasse verwendet).

Dabei handelt es sich um Methoden, nur selten auch um Felder, die die Kernaufgabe Ihrer Klasse widerspiegeln. Um das musikalische Beispiel fortzusetzen: Wenn Sie nach `Song` als Nächstes eine Klasse `Playlist` implementieren, dann wären die Methoden, um einen Song hinzuzufügen, den aktuellen Song auszulesen und zum nächsten oder vorigen Song zu springen, ein Teil der API von `Playlist`, denn durch diese Methoden steuern Sie die Playlist in ihrer Kernaufgabe.

Am anderen Ende der Skala benutzen Sie `private`-Sichtbarkeit für alles, was nur intern von der Klasse benötigt wird. Dies betrifft Implementierungsdetails, die außerhalb der Klasse nicht von Interesse sind. Die Klasse `Song` kann in ihrer öffentlichen API die Methode `play()` enthalten. Dazu muss sie intern eine Datei öffnen, den Inhalt der Datei decodieren und mehr. Diese Details interessieren die Außenwelt aber nicht, die Methoden `oeffneDatei()` und `decodiere()` können `private` sein. Außerdem ist es in Java ein verbreitetes Muster, alle Felder `private` zu deklarieren und Zugriff auf sie nur durch Methoden mit breiterer Sichtbarkeit zu erlauben (siehe Abschnitt 5.4.5, »Zugriffsmethoden«).

`protected` wird verwendet, um das Verhalten einer Oberklasse in Unterklassen zu verändern, indem sie dort *überschieben* werden. Mehr dazu finden Sie in Kapitel 6, »Objektorientierung«, unter den Stichwörtern *Vererbung* und *Polymorphie*.

Package-Zugriff verwenden Sie dann, wenn mehrere Klassen in einem Package eng zusammenarbeiten, um ihre Aufgabe zu erfüllen. In diesem Fall kann es sinnvoll sein, dass sie direkten Zugriff auf Implementierungsdetails der jeweils anderen Klassen im Package haben, die aber nicht für den Rest der Welt zugänglich sein sollen.

5.3 Felder

Ein Objekt kann zusammengehörige Daten und Operationen gruppieren, aber es muss nicht notwendigerweise beides beinhalten. Ein Objekt kann auch ein reiner Datencontainer sein. Um solche Objekte zu erzeugen, deklarieren Sie in Ihrer Klasse lediglich Felder, keine Methoden, und greifen von außen auf diese Felder zu.

5.3.1 Felder deklarieren

Die Deklaration eines Feldes unterscheidet sich nicht wesentlich von der einer lokalen Variablen. Genau wie dort geben Sie einen Datentyp und einen Namen an. Anders als bei lokalen Variablen können Sie für Felder aber auch einen Access Modifier angeben.

```
public class Song {
    public String titel;
    public String interpret;
    public int laengeInSekunden;
}
```

Listing 5.3 Eine reine Datenklasse für die Musiksammlung

In einer Klasse ohne Operationen, wie es `Song` momentan noch ist, ist es natürlich nicht sinnvoll, die Felder `private` zu deklarieren. In diesem Fall sind die Felder `public`, Zugriff ist von überall erlaubt.

Ein weiterer Unterschied zwischen Feldern und lokalen Variablen ist, dass Felder einen *Default-Wert* erhalten. Sie müssen deshalb nicht jedes Feld explizit auf `null`, `0` oder `false` setzen, bevor Sie damit arbeiten.

5.3.2 Zugriff auf Felder

Um von außerhalb des Objekts auf Felder (oder Methoden) zuzugreifen, verwenden Sie den Punktoperator. Der Zugriff funktioniert damit genauso wie auf eine lokale Variable.

```
Song cupOfJava = new Song();
cupOfJava.title = "Cup of Java";
cupOfJava.interpret = "Compilaz";
cupOfJava.laengeInSekunden = 217;
if (cupOfJava.laengeInSekunden > 180){
    …
}
```

Listing 5.4 Zugriff auf Felder

5.4 Methoden

Objekte, die Daten thematisch gruppieren, haben eine Existenzberechtigung, aber was die Objektorientierung ausmacht, ist die Zusammenfassung der Daten mit den dazugehörigen Operationen. Reine Datenstrukturen gab es bereits vorher. Erst mit Methoden wird es wirklich objektorientierte Programmierung.

Sie haben nun bereits einige Methoden gesehen, es gab sie in jeder Übung. Es wird Zeit, sie detaillierter zu betrachten. Eine Methode ist eine Abfolge von Instruktionen, die unter dem Methodennamen zusammengefasst werden. So können Sie diese spezielle Instruktionsfolge an jeder beliebigen Stelle Ihres Programms aufrufen, ohne sie wiederholen zu müssen.

Eine Methode kann einen *Rückgabewert* haben, ein Ergebnis, das an den Aufrufer der Methode weitergegeben wird. Außerdem kann eine Methode einen oder mehrere *Parameter* haben, Werte die der Methode beim Aufruf übergeben werden und die sie zur Erfüllung ihrer Aufgabe benötigt.

In der Deklaration einer Methode müssen alle diese Eigenschaften auftauchen: der Typ des Rückgabewertes, der Methodenname, Typen und Namen der Parameter (soweit vorhanden) und ein optionaler Access Modifier. Ein kleiner Stolperstein ist dabei, dass auch eine Methode, die keinen Rückgabewert hat, den Typ ihres Rückgabewertes deklarieren muss. Was zunächst paradox klingt, ist in der Praxis leicht umgesetzt: Wenn eine Methode keinen Rückgabewert hat, dann wird der Typ ihres Rückgabewertes mit dem Schlüsselwort `void` deklariert. Eine Methodendeklaration sieht damit beispielsweise so aus:

```
public void play(){
    …
}
```

Listing 5.5 Methodendeklaration

Die Deklaration beginnt mit dem Access Modifier, gefolgt von Rückgabewert, Methodenname und der Parameterliste in Klammern. Selbst wenn die Methode keine Parameter annimmt, müssen Sie das leere Klammerpaar schreiben. In geschweiften Klammern folgt dann der Methodenrumpf, die Liste von Instruktionen, die die Methode ausführen soll.

Ich muss Sie übrigens insofern enttäuschen, als Sie in diesem Buch keine Musikdateien abspielen werden, sondern lediglich Ihre Musiksammlung verwalten.

Felder, lokale Variablen und »this«

Sie haben bereits gesehen, wie Sie mit dem Punktoperator von außerhalb des Objekts auf seine Felder zugreifen. Innerhalb des Objekts brauchen Sie den Punktoperator

nicht, Sie können auf Felder einfach mit deren Namen zugreifen, genau wie auf lokale Variablen. Das hat aber einen kleinen Nachteil: Wenn es eine lokale Variable desselben Namens gibt, dann können Sie so nicht mehr auf das Feld zugreifen, es wird immer die lokale Variable verwendet. Man sagt, die lokale Variable *verdeckt* das Feld. Im Englischen spricht man von *Variable Shadowing*.

```
public class ImSchatten {
    public int zahl = 11;

    public void rechne(){
        int zahl = 17;
        System.out.println("Zahl: " + zahl);
    }
}
```

Listing 5.6 Eine verdeckte Variable

Aber auch verdeckte Felder sind nicht komplett unzugänglich. Um auf Sie zuzugreifen, können Sie das Schlüsselwort `this` verwenden. Es verhält sich genau wie jede andere Objektreferenz, aber es referenziert immer das Objekt, in dem Sie sich gerade befinden. Insbesondere erlaubt es den Zugriff auf alle Felder oder Methoden des Objekts, auch wenn Felder von lokalen Variablen verdeckt werden.

```
public class AusDemSchatten {
    public int zahl = 11;

    public void rechne(){
        int zahl = 17;
        System.out.println("Zahl (lokal): " + zahl);
        System.out.println("Zahl (Feld): " + this.zahl);
    }
}
```

Listing 5.7 Eine verdeckte Variable

Manche Java-Programmierer sind der Meinung, dass der Zugriff auf ein Feld immer über `this` erfolgen sollte, weil so auf den ersten Blick sichtbar ist, dass auf ein Feld zugegriffen wird und nicht auf eine lokale Variable. Da aber jede IDE diesen Unterschied hervorhebt – Netbeans schreibt zum Beispiel Feldnamen in Grün und lokale Variablen in Schwarz –, ist dies nicht mehr so wichtig, wie es vielleicht einmal war. In diesem Punkt gibt es keine allgemein anerkannte Empfehlung.

Der Zugriff auf verdeckte Felder ist zwar ein wichtiger Zweck von `this`, aber nicht der einzige. `this` verhält sich wirklich in jeder Hinsicht so wie eine Objektvariable des

richtigen Typs, und Sie können es auch als Parameter an eine Methode übergeben oder als Rückgabewert einer Methode verwenden.

5.4.1 Übung: Eine erste Methode

Schreiben Sie nun Ihre erste Methode. Beginnen Sie mit der Klasse `Song`, wie abgedruckt, mit zwei String-Feldern `interpret` und `titel` und einem `int`-Feld `laengeInSekunden`. Fügen Sie eine Methode `drucke` hinzu, die die Daten des Songs auf die Kommandozeile ausgibt.

Schreiben Sie dann eine Klasse `Musicplayer`, die ein `Song`-Objekt erzeugt, seine Felder mit Werten befüllt und es schließlich mit Hilfe Ihrer Methode ausgibt. Die Lösung zu dieser Übung finden Sie im Anhang.

5.4.2 Rückgabewerte

Eine Methode kann ihrem Aufrufer ein Ergebnis zurückgeben, man spricht von einem *Rückgabewert*. Um einen Wert zurückzugeben, müssen Sie den Typ des Rückgabewertes in der Methodendeklaration angeben.

Die Beispiele oben haben für ihren Rückgabewert den Typ `void` verwendet, was so viel bedeutet wie »kein Rückgabewert«. An derselben Stelle können Sie aber jeden beliebigen Typ angeben, den Ihre Methode zurückgibt.

```
public class Song {
    …
    public String toString(){
        …
    }
}
```

Listing 5.8 Eine Methode mit Rückgabewert

Die Methode `toString` liefert ihrem Aufrufer einen `String`. Einen Wert zurückgeben kann die Methode mit dem Schlüsselwort `return`, gefolgt von dem Wert, der zurückgegeben werden soll. Die Ausführung der Methode wird sofort unterbrochen, und die aufrufende Methode wird mit dem Rückgabewert fortgesetzt.

Die »toString«-Methode

Es handelt sich bei `toString` übrigens um eine besondere Methode, denn genau diese Methode wird von der JVM gerufen, wenn sie eine textuelle Repräsentation eines Objekts benötigt, zum Beispiel um es mit `System.out.println()` auf die Kommandozeile zu schreiben.

Anweisungen, die in der Methode nach dem `return` stehen, werden nicht ausgeführt. Wenn eine Methode dadurch Anweisungen enthält, die niemals zur Ausführung kommen können, dann führt das zu einem Compiler-Fehler.

```
public String machEinenString(){
    return "Ein String";
    System.out.println("Nach dem return");
}
```

Listing 5.9 Diese Methode wird nicht kompilieren.

»return« und »void«-Methoden

Auch in `void`-Methoden können Sie `return` verwenden, hier allerdings ohne einen Wert. Sie können so zwar keinen Wert zurückgeben, aber die Ausführung der Methode wird dennoch unterbrochen. So können Sie eine Methode abbrechen, bevor ihr Ende erreicht ist, aber ohne einen Fehler zu werfen. Man spricht von einem *Early Return*. Am Ende einer `void`-Methode muss niemals ein `return` stehen, sie endet hier auch von selbst.

Der Aufrufer der Methode kann nicht unterscheiden, ob die Methode bis zum Ende gelaufen ist oder schon vorher von einem `return` unterbrochen wurde.

Wenn eine Methode einen Rückgabewert deklariert, dann **muss** sie auch einen Wert zurückliefern. Es ist nicht zulässig, in einer Methode mit Rückgabewert das `return`-Statement auszulassen. Ist ein Rückgabewert deklariert, dann müssen Sie auch einen Wert zurückgeben. Sie können aber bei Methoden, die ein Objekt zurückgeben, mit `return null` einen `null`-Wert liefern. Im Beispiel `Musicplayer` könnte das der Fall sein, wenn Sie den sechsten Titel aus einer Playlist mit nur fünf Einträgen auslesen wollen, sie geben `null` zurück mit der Bedeutung »dieser Eintrag existiert nicht«.

[!]

Referenzen!

Auch bei Rückgabewerten gilt, dass Sie bei Objekttypen grundsätzlich mit Referenzen arbeiten. Wenn sie also ein Objekt zurückgeben, das auch in einem Feld gespeichert ist, dann erhält der Aufrufer eine Referenz auf dasselbe Objekt. Die Konsequenzen sind dieselben, die Sie bereits beobachtet haben, wenn zwei Variablen dasselbe Objekt referenzieren: Änderungen, die Sie durch eine Referenz machen, sind durch die andere Referenz sichtbar, weil hinter den zwei Referenzen nur ein Objekt steht.

5.4.3 Übung: Jetzt mit Rückgabewerten

Implementieren Sie in der Klasse `Song` eine Methode mit dem Namen `formatiereZeit` und dem Rückgabetyp `String`. Diese Methode soll aus dem Feld `laengeInSekunden` eine Zeitangabe im Format Minuten:Sekunden berechnen und diese zurückgeben.

Implementieren Sie außerdem die oben angesprochene `toString`-Methode für die Klasse `Song`. Der `String`, den sie zurückgibt, soll alle wichtigen Felder des Songs enthalten: Titel, Interpret und die formatierte Zeit. Dann passen Sie die Methode `drucke` an, so dass sie die `toString`-Methode benutzt, anstatt ihren Ausgabe-`String` selbst zusammenzusetzen. Die Lösung zu dieser Übung finden Sie im Anhang.

5.4.4 Parameter

Genau wie ein Rückgabewert das ist, was aus einer Methode herauskommt, ist ein Parameter das, was Sie in eine Methode hineingeben. Wenn Sie Addition als Methode definieren wollen, dann sind die Summanden die Parameter.

Eine Methode kann beliebig viele Parameter haben, die jeweils mit Typ und Namen in den Klammern der Methodendeklaration angegeben werden. Mehrere Parameter werden durch Kommata getrennt:

```
public int summe(int summand1, int summand2){…}
```

Listing 5.10 Methodendeklaration

Innerhalb der Methode verhalten sich Parameter genau wie eine lokale Variable. Sie können ihre Werte auf die gleiche Art auslesen.

```
public int summe(int summand1, int summand2){
    return summand1 + summand2;
}
```

Listing 5.11 Verwendung von Parametern

Sie können Parametern auch neue Werte zuweisen. Diese neuen Werte werden **nicht** nach außen weitergegeben. Wenn Sie allerdings Objekte als Parameter akzeptieren und die Felder dieser Objekte ändern, dann sind diese Änderungen sehr wohl außerhalb der Methode sichtbar. Auch als Parameter werden bei Objekten nur Referenzen übergeben, der Fachbegriff dafür ist *Pass by Reference*.

Erst durch Parameter werden Ihre Methoden wirklich wiederverwendbar. Was wäre der Sinn einer Additionsmethode ohne Parameter? Sie könnte immer nur die zwei gleichen, konstanten Zahlen addieren. Oder denken Sie an eine `Playlist`-Klasse für Ihren Musikspieler. Sie können ihr eine Methode `fuegeHinzu` geben, eine Methode `entferne`, aber was würden Sie hinzufügen und entfernen, wenn Sie keine Parameter übergeben könnten?

Methodensignaturen

Eine Methode wird in Java an zwei Merkmalen identifiziert: dem Methodennamen und der Parameterliste. Diese Kombination nennt man die *Signatur* der Methode.

Das bedeutet insbesondere, dass es in Java möglich ist, in einer Klasse mehrere Methoden mit demselben Namen zu deklarieren, solange sich ihre Parametertypen unterscheiden. Es müssen die Parametertypen sein, die sich unterscheiden, nicht die Parameternamen, denn nur anhand der Typen kann der Compiler unterscheiden, welche Methode mit einem Aufruf gemeint ist.

Der Rückgabewert ist **nicht** Teil der Methodensignatur, und entsprechend ist es auch nicht erlaubt, dass sich zwei Methoden nur in ihrem Rückgabewert unterscheiden. Auch das liegt daran, dass der Compiler Aufrufe dieser Methoden nicht unterscheiden könnte.

Wenn sich mehrere Methoden einer Klasse nur durch ihre Parametertypen unterscheiden, spricht man von *überladenen Methoden* (*Method Overloading*).

Dieses Beispiel mit überladenen Methoden würde kompilieren:

```
public class Playlist{
    public void fuegeHinzu (Song einSong){…}
    public void fuegeHinzu (Song einSong, int anStelle){…}
    public void fuegeHinzu (List songs){…}
}
```

Listing 5.12 »Playlist« mit überladenen Methoden

Dieses Beispiel zeigt den üblichen Zweck von überladenen Methoden: Mehrere Methoden erfüllen dieselbe Aufgabe, aber mit unterschiedlichen Parametern. `fuegeHinzu(Song)` würde einen `Song` am Ende der Playlist hinzufügen, `fuegeHinzu(Song, int)` ihn an einer beliebigen Stelle in der Liste einfügen und `fuegeHinzu(List)` mehrere Songs auf einmal hinzufügen.

Dieses Beispiel hingegen ist nicht funktionsfähig:

```
public class Playlist{
    public int groesse(){…}
    public long groesse(){…}
}
```

Listing 5.13 Fehlerhaftes Method Overloading

Die beiden Methoden unterscheiden sich nur durch den Rückgabewert. Das quittiert der Compiler mit der Fehlermeldung: »Method groesse is already defined in class.«

5.4.5 Zugriffsmethoden

Es ist eine in Java weit verbreitete Praxis, Felder eines Objekts nicht nach außen sichtbar zu machen. Alle Felder werden nach diesem Muster `private` deklariert, und Zugriff auf sie wird nur durch *Zugriffsmethoden* ermöglicht.

```
public class Song {
    private int laengeInSekunden;

    public int getLaengeInSekunden(){
        return laengeInSekunden;
    }

    public void setLaengeInSekunden (int laengeInSekunden){
        this.laengeInSekunden = laengeInSekunden;
    }
}
```

Listing 5.14 Zugriffsmethoden

So wird ein Feld durch Zugriffsmethoden vor der Außenwelt verborgen. Selbst wenn Sie, wie es in diesem Buch der Fall ist, Feld- und Methodennamen auf Deutsch vergeben, dann sollten die Zugriffsmethoden trotzdem `set…` zum Schreiben und `get…` zum Lesen heißen. Die einzige Ausnahme sind `boolean`-Felder, die Methode für lesenden Zugriff heißt bei ihnen `is…`.

Diese Konvention wird von vielen Frameworks und Komponenten, auch in der Standardbibliothek, vorausgesetzt, um auf Eigenschaften zuzugreifen. Wenn Sie versuchen, diese Präfixe einzudeutschen, können diese Komponenten nicht mehr mit Ihren Klassen arbeiten. Weil die Präfixe so festgesetzt sind, spricht man häufig auch von *Gettern* und *Settern*.

Aber wofür soll das gut sein, warum sollte man nicht von außen direkt auf die Felder zugreifen? Man möchte so die öffentliche API einer Klasse von ihren Implementierungsdetails abstrahieren (*Datenkapselung*, *Encapsulation* oder *Implementation Hiding*). So möchten Sie zum Beispiel, dass die Klasse `Song` nach außen hin die Eigenschaft `laengeInSekunden` besitzt. Aber Sie müssen deswegen nicht unbedingt die Sekundenzahl in einem Feld speichern, vielleicht gibt es gute Gründe dafür, stattdessen die Länge in Millisekunden als einen `long`-Wert zu speichern oder ein Objekt vom Typ `java.time.Duration`. Durch Zugriffsmethoden können Sie den intern gespeicherten Datentyp von der Außendarstellung trennen.

```
import java.time.Duration;
public class Song {
    private Duration laenge;
```

```
    public int getLaengeInSekunden(){
        return (int) laenge.getSeconds();
    }

    public void setLaengeInSekunden (int laengeInSekunden){
        this.laengeInSekunden = Duration.ofSeconds(laengeInSekunden);
    }
}
```

Listing 5.15 Trennung von API und Implementierung

Nach außen hin hat sich gegenüber dem vorherigen Beispiel nichts geändert, aber innerhalb von `Song` verwenden Sie jetzt die `Duration`-Klasse, um die Dauer des Stücks zu speichern. Würden Sie ohne Zugriffsmethoden arbeiten und anderen Klassen direkten Zugriff auf die Felder gewähren, dann müssten jetzt alle Klassen angepasst werden, die `Song` verwenden. Das ist schon ärgerlich, wenn Sie die Klasse nur in Ihrem eigenen Projekt verwenden. Ist die Klasse Bestandteil einer Bibliothek, die auch von anderen eingesetzt wird, dann macht Ihnen eine solche Änderung nur wenige Freunde. Mit Zugriffsmethoden passiert das nicht, die öffentliche API bleibt unverändert, andere Programmierer, die Ihre Klassen verwenden, bleiben glücklich, und was Sie in Ihrer Klasse tun, ist Ihre eigene Sache.

Ein weiterer Vorteil von Zugriffsmethoden ist, dass Sie neue Werte, die durch einen Setter gesetzt werden, *validieren* können, also Einschränkungen für die gesetzten Werte erzwingen können, die über den Datentyp hinausgehen. Sie können verhindern, dass für einen Objekttyp `null` gesetzt wird, für einen Zahlentyp ein Wert außerhalb des zulässigen Bereichs oder was auch immer sonst in Ihrer Anwendung sinnvoll ist. Wenn Sie direkten Zugriff auf die Felder zulassen, haben Sie diese Möglichkeit nicht.

Wird ein ungültiger Wert übergeben, haben Sie zwei Möglichkeiten, damit umzugehen. Sie können den Wert ignorieren und nicht ins Feld speichern. Diese Lösung ist zwar einfach, aber unschön, denn der Aufrufer enthält keinen Hinweis darauf, dass der Wert ungültig war, und wird überrascht, wenn das Objekt nach wie vor den alten Wert enthält. Die bessere Lösung ist, einen Fehler zu werfen. Dies ist ein kleiner Vorgriff auf Kapitel 9, »Fehler und Ausnahmen«, wo wir uns ausführlich mit Fehlern beschäftigen werden.

```
public class Song {
    private String interpret;
    public String getInterpret(){
        return this.interpret;
    }
    public void setInterpret(String interpret){
```

```
        if (interpret == null){
            throw new IllegalArgumentException("Interpret ist null");
        }
        this.interpret = interpret;
    }
}
```

Listing 5.16 Validierung im Setter

Wie Sie im Beispiel sehen, wird ein Fehler mit der `throw`-Anweisung geworfen. Sie unterbricht die Methode sofort und gibt den Fehler an die aufrufende Methode weiter. Die spezielle Fehlerart `IllegalArgumentException` ist genau für den Fall gedacht, dass ein ungültiger Wert als Parameter übergeben wurde. Der Text `"Interpret ist null"` hat keine Bedeutung im Programm, sondern ist rein informativ, er klärt lediglich darüber auf, was genau schiefgegangen ist. Momentan können Sie diesen Fehler noch nicht behandeln, das lernen Sie in Kapitel 9, »Fehler und Ausnahmen«. Wenn der Fehler also geworfen wird, bricht das Programm sofort mit einer Fehlermeldung ab.

Als letzten hier erwähnenswerten Vorteil von Zugriffsmethoden können Sie so Eigenschaften implementieren, die von außen nur gelesen werden können, aber nicht geschrieben. Dazu implementieren Sie einfach keinen Setter oder geben auch dem Setter eine eingeschränkte Sichtbarkeit. Andersherum können Sie natürlich auch Eigenschaften von außen nur schreibbar machen, dafür ist es allerdings schwieriger, eine sinnvolle Anwendung zu finden.

5.4.6 Übung: Zugriffsmethoden

Stellen Sie jetzt Ihre `Song`-Klasse auf Zugriffsmethoden um. Ändern Sie dafür alle Felder auf `private`-Sichtbarkeit, schreiben Sie dann die dazu passenden Zugriffsmethoden.

Alle Setter sollen die gesetzten Werte validieren: Titel und Interpret dürfen nicht `null` sein, für die Länge darf kein negativer Wert gesetzt werden. Wenn falsche Werte übergeben werden, werfen Sie, wie oben gezeigt, einen Fehler.

Zugriffsmethoden erlauben es Ihnen nicht nur, die öffentliche API vom internen Datenformat unabhängig zu halten, sondern auch, ein Feld in mehreren verschiedenen Formaten zugänglich zu machen. Schreiben Sie, zusätzlich zu den vorhandenen Gettern und Settern, eine Methode `setLaenge`, die die drei Parameter `stunden`, `minuten` und `sekunden` erwartet und aus ihnen die neue Länge berechnet und setzt. Schreiben Sie drei Methoden `getStunden`, `getMinuten` und `getSekunden`, die den jeweiligen Teil der Länge zurückgeben. Da Sie jetzt eine Methode anbieten, in der explizit Stunden gesetzt werden können, sollten diese auch in der Ausgabe auftauchen. Erweitern Sie die Methode `formatiereZeit`, so dass sie die Stunden angibt, wenn der Song eine Stunde lang oder länger ist. Die Lösung zu dieser Übung finden Sie im Anhang.

5.5 Warum Objektorientierung?

Bevor wir tiefer in die Objektorientierung einsteigen, möchte ich eine Frage beantworten, die viele von Ihnen sich gerade stellen. Was ist eigentlich so toll an Objektorientierung? Es gab auch schon komplexe Programme, bevor die objektorientierte Programmierung das verbreitetste Programmierparadigma wurde, und auch heute ist objektorientiert nicht die einzige Art zu programmieren. Warum also objektorientierte Programmierung und nicht prozedurale, funktionale, oder sonstige?

Einer der wichtigsten Gründe hat interessanterweise nichts mit dem Computer zu tun, sondern mit dem Menschen. In Objekten zu denken ist für uns von frühester Kindheit an natürlich. Wir nehmen die Welt nicht als eine Menge von abstrakten, unzusammenhängenden Daten war, sondern als Objekte, die etwas sind und etwas tun (die Eigenschaften und Operationen haben) und miteinander interagieren. Schon bei einem einfachen Beispiel wie einem Fußball ist es uns gar nicht möglich, ihn *nicht* als ein Objekt zu betrachten. Natürlich können Sie versuchen, die Eigenschaften und Fähigkeiten des Balls abstrakt aufzuzählen, ohne dabei an einen Ball zu denken: »etwas Kugelrundes«, »etwas aus Leder«, »etwas mit Luft Gefülltes«, »etwas, das hüpfen kann«. Aber machen Sie das Experiment: Nehmen Sie einen Freund zur Seite, lesen Sie ihm diese Liste vor, und ich bin beinahe sicher, Ihr Freund wird sagen »Ein Ball!«, bevor Sie das Ende der Liste erreichen. Denn wir können nicht anders, als in Objekten zu denken.

In Objekten zu programmieren ist deshalb eine Metapher, die unserem natürlichen Denken angepasst ist und es uns so einfach macht, über Programme nachzudenken. Auch wenn auf technischer Ebene mit Zeigern, Speicheradressen, Nullen und Einsen gearbeitet wird, die Objektmetapher macht das Programm verständlich.

Aber Neuropsychologie ist nicht der einzige Vorteil der Objektorientierung, es gibt auch greifbarere Gründe. So gruppieren Objekte zusammengehörige Daten auf eine Art, die sie nicht durcheinandergeraten lässt: Titel und Interpret sind in einem `Song`-Objekt gespeichert, es ist so gut wie unmöglich, dass ein Titel plötzlich versehentlich einem anderen Interpreten zugeordnet wird.

Objekte sind leicht wiederverwendbar. Wenn Sie ein weiteres Projekt entwickeln, in dem es sich um Musik dreht, können Sie die Klassen `Song`, `Playlist` usw. sehr leicht in eine Bibliothek auslagern, die sowohl von dem hier entwickelten Projekt wie auch dem neuen Projekt genutzt wird. Sie müssen die Klassen nicht kopieren oder gar neu entwickeln. Und wenn Sie für ein Projekt eine neue Methode hinzufügen, steht diese sofort überall zur Verfügung, wo Sie die Klasse bereits einsetzen.

Objekte bewahren Sie vor doppeltem Code. Eine Methode, die mit den Daten eines Objekts arbeitet, wird in dessen Klasse implementiert und steht damit überall zur Verfügung. Sie müssen Code nicht kopieren, nur weil Sie an einer anderen Stelle des Programms dieselbe Funktionalität benötigen. Überall, wo Sie die Daten haben, haben Sie auch die Methode, die mit ihnen arbeitet.

Zusammen bringen diese Vorteile eine enorme Produktivitätssteigerung, die durch ein weiteres sehr wichtiges Merkmal der Objektorientierung noch größer ausfällt. Klassen können Funktionalität von anderen Klassen übernehmen und erweitern, indem sie von ihnen *erben*.

Wie Sie Ihre Klassen von anderen Klassen erben lassen, lernen Sie ausführlich in Kapitel 6, »Objektorientierung«, aber warum das eine tolle Idee ist, ist schon hier interessant. Vererbung macht Klassen noch wiederverwendbarer, als sie es sowieso schon sind. Indem Sie eine Klasse erweitern, also von ihr erben, können Sie alle Möglichkeiten nutzen, die die Oberklasse bietet, und sie zusätzlich um ihre eigenen erweitern. Ein Beispiel:

```
public class KaraokeSong extends Song {
    private String text;
    public String getText(){…}
    public void setText(String text){…}
}
```

Listing 5.17 »Song« erweitert

Ein Objekt vom Typ `KaraokeSong` ist in jeder Beziehung ein `Song`, es kann überall verwendet werden, wo ein normaler `Song` erwartet wird, aber zusätzlich können Sie auf den Liedtext zugreifen. Sie müssen es mir nicht jetzt sofort glauben, aber ich bin sicher, Sie werden zum Ende dieses Buches überzeugt sein, dass Vererbung großartig ist.

Eine Kleinigkeit zum Thema Vererbung habe ich Ihnen übrigens verschwiegen: Auch wenn Ihre Klassen nicht explizit mit `extends` von einer anderen Klasse erben, haben sie dennoch *immer* eine Oberklasse. Diese Oberklasse heißt `Object`, und sie steht für jedes Java-Objekt an oberster Stelle des Stammbaums.

`Object` kann nicht viel, aber was es kann, steht durch die Vererbungsbeziehung jeder Ihrer Klassen zur Verfügung. Eine Methode von `Object` haben Sie in diesem Kapitel bereits kennengelernt: `toString` ist ursprünglich in `Object` implementiert, und wenn Ihre Klasse die Methode nicht überschreibt, also eine eigene Methode mit derselben Signatur deklariert, dann erbt sie die `toString`-Methode von `Object`. Für `Song` sähe die Ausgabe dann so aus:

```
de.kaiguenster.javaintro.music.Song@106d69c
```

5.6 Konstruktoren

Eine weitere Art von Methode ist wichtig, um Objekte zu deklarieren: *Konstruktoren*. Wie der Name, mit etwas Fantasie, verrät, ist es deren Aufgabe, neue Objekte zu konstruieren. Konstruktoren initialisieren ein neues Objekt, sie setzen Anfangswerte für Felder, bereiten Ressourcen vor, die das Objekt benötigen wird, und sorgen dafür, dass ein neues Objekt sofort in einem validen Zustand ist.

Nehmen Sie als Beispiel einmal mehr die Klasse `Song`, die Sie schon durch das ganze Kapitel begleitet. Die Setter für Titel und Interpret erlauben es nicht, einen `null`-Wert für diese Felder zu setzen. `null` ist für diese Felder ein ungültiger Wert. Wenn Sie aber mit `new Song()` ein neues Objekt vom Typ `Song` erzeugen, dann haben die Felder zunächst den Wert `null`, und das Objekt ist damit in einem ungültigen Zustand. Das lässt sich zum Beispiel mit einem Konstruktor verhindern.

5.6.1 Konstruktoren deklarieren und aufrufen

Sie deklarieren einen Konstruktor wie jede andere Methode auch, mit zwei Abweichungen. Zum einen geben Sie für einen Konstruktor niemals einen Rückgabewert an, auch nicht `void`, diese Stelle in der Deklaration entfällt. Zum anderen trägt ein Konstruktor immer den Namen der Klasse, zu der er gehört. Insbesondere gilt das auch für Groß- und Kleinschreibung. Der Name eines Konstruktors beginnt entgegen der Namenskonvention für andere Methoden mit einem Großbuchstaben, weil der Klassenname mit einem Großbuchstaben beginnt.

```
public class Song {
    public Song(){
        this.titel = "";
        this.interpret = "";
    }
    …
}
```

Listing 5.18 Ein Konstruktor für »Song«

Dieser Konstruktor setzt die Felder `titel` und `interpret` auf gültige Anfangswerte. Wie der Konstruktor aufgerufen wird, haben Sie vielleicht schon erraten, nachdem Sie nun die Deklaration gesehen haben. Der Konstruktoraufruf gehört zum Schlüsselwort `new`, so wie in `new Song()`. Das sieht so aus, als habe es den gezeigten Konstruktor schon gegeben, bevor Sie ihn deklariert haben. Und es sieht nicht nur so aus. Wenn in einer Klasse kein Konstruktor deklariert wird, dann wird von Java automatisch ein Konstruktor ohne Parameter erzeugt. Deshalb heißt der Konstruktor ohne Parameter auch *Default-Konstruktor*.

Es ist häufig sinnvoll, einen Default-Konstruktor anzubieten, der vom Objekt benötigte Ressourcen vorbereitet, zum Beispiel Netzwerkverbindungen öffnet oder Dateiinhalte einliest. Im gezeigten Fall ist der Default-Konstruktor allerdings nicht notwendig, derselbe Effekt lässt sich auch so erreichen:

```
public class Song {
    private String titel = "";
    private String interpret = "";
    …
}
```

Listing 5.19 Default-Werte ohne Konstruktor

Dieser Code hat den gleichen Effekt; nur um Default-Werte zu setzen, brauchen Sie keinen Default-Konstruktor. Außerdem ist ein Leer-String nicht wirklich sinnvoller als `null`. Er vermeidet zwar eine `NullPointerException`, aber inhaltlich verbessert er die Situation kaum, ein neuer Song hat immer noch keinen Titel oder Interpreten.

Aber wie anderen Methoden auch können Sie einem Konstruktor Parameter übergeben, damit lässt sich sicherstellen, dass auch ein ganz frischer `Song` richtig initialisiert wird.

```
public class Song {
    public Song(String titel, String interpret, int laengeInSekunden){
        this.setTitel(titel);
        this.setInterpret(interpret);
        this.setLaengeInSekunden(laengeInSekunden);
    }
}
```

Listing 5.20 Konstruktor mit Parametern

Nun können Sie, wenn Sie einen neuen Song erstellen, die Feldwerte gleich mitgeben:

```
new Song("Iffy and the Elses", "Decide!", 345);
```

Genau genommen können Sie jetzt nicht nur die Feldwerte mitgeben, Sie müssen es. Ein Default-Konstruktor wird wirklich nur dann automatisch erzeugt, wenn die Klasse *keinen* Konstruktor deklariert. Sobald es einen gibt, egal, ob mit oder ohne Parameter, wird kein Konstruktor mehr automatisch erzeugt. Möchten Sie, dass der Default-Konstruktor auch weiterhin existiert, dann müssen Sie ihn selbst deklarieren.

Sie sehen, dass der Konstruktor im Beispiel nicht direkt auf die Felder zugreift, sondern die Zugriffsmethoden verwendet. Der Konstruktor könnte natürlich direkt auf die `private`-Felder zugreifen, aber durch die Zugriffsmethoden wird schon sichergestellt, dass keine `null`-Werte gesetzt werden können. Diese Validierung machen wir

uns auch im Konstruktor zunutze, denn das Ziel war es schließlich, alle Felder von Anfang an mit gültigen Werten befüllt zu haben.

Mehrere Konstruktoren für eine Klasse

Sie sind für Ihre Klassen nicht auf einen einzigen Konstruktor beschränkt. Sie können für eine Klasse mehrere Konstruktoren anbieten, und sehr viele Klassen des JDKs tun das ebenfalls. Genau wie bei Methoden gilt auch bei Konstruktoren, dass Konstruktoren einer Klasse unterschiedliche Signaturen haben müssen. Der Name eines Konstruktors ist aber vorgeschrieben, alle Konstruktoren müssen sich daher in ihrer Parameterliste unterscheiden.

Es gibt zwei gute Gründe für eine Klasse, mehrere Konstruktoren zu deklarieren. Zum einen können Sie, wie Sie es auch schon bei Zugriffsmethoden gesehen haben, Daten in verschiedenen Formaten entgegennehmen, zum Beispiel die Länge als Sekundenzahl in einem `int`-Parameter oder in einem `Duration`-Objekt. Sie können aber auch mehrere Konstruktoren deklarieren, um Default-Werte für Ihre Felder vorzugeben. So können Sie in der Klasse `Song`, falls kein Interpret angegeben wurde, den Default-Wert »Unbekannter Künstler« in das Feld schreiben. Das ginge zwar auch mit einem einfachen `if (interpret == null)` im Konstruktor, aber die Klasse wird dadurch schwerer zu benutzen: Wenn Sie den Konstruktor aufrufen, müssen Sie nachdenken oder im Javadoc nachlesen, ob `null` ein erlaubter Wert für den Parameter `interpret` ist. Steht ein Konstruktor mit der Signatur `public Song(String titel, int laenge)` zur Verfügung, dann stellt sich diese Frage gar nicht erst.

Wenn Sie mehrere Konstruktoren verwenden, um verschiedene Datenformate zu akzeptieren oder um Default-Werte zu setzen, dann sollten Sie nicht jeden Konstruktor ausimplementieren, sondern einen weiteren Konstruktor mit dem Default-Wert oder dem transformierten Datenformat aufrufen. Um aus einem Konstruktor einen anderen Konstruktor derselben Klasse aufzurufen, verwenden Sie das Schlüsselwort `this` als Methode.

```
public Song(String titel, String interpret, int laengeInSekunden) {
    this.setTitel(titel);
    this.setInterpret(interpret);
    this.setLaengeInSekunden(laengeInSekunden);
}

public Song(String titel, int laengeInSekunden) {
    this(titel, "Unbekannter Künstler", laengeInSekunden);
}
```

Listing 5.21 Constructor Chaining

Diese Technik heißt *Constructor Chaining* und dient einmal mehr dazu, doppelten Code zu vermeiden. Jeder Konstruktor erfüllt genau eine Aufgabe, er setzt einen Default-Wert oder wandelt Daten in ein anderes Format um und leitet dann an einen anderen Konstruktor weiter. Im Idealfall gibt es nur einen Konstruktor, der am Ende wirklich Daten in die Felder schreibt. Wenn Sie `this` aufrufen, dann muss dieser Aufruf die erste Anweisung des Konstruktors sein. Das führt zu der etwas abstrusen Situation, dass der oben gezeigte Konstruktor kompiliert, dieser hier aber nicht:

```
public Song(String titel, int laengeInSekunden) {
    String defaultKuenstler = "Unbekannter Künstler";
    this(titel, defaultKuenstler, laengeInSekunden);
}
```

Listing 5.22 Dieser Konstruktor kompiliert nicht.

Diese Einschränkung macht es zwar manchmal schwieriger, Konstruktoren sinnvoll zu verketten, aber sie gibt es in Java von Anfang an und wird sich wohl auch in Zukunft nicht ändern.

Nicht öffentliche Konstruktoren

Auch bei Konstruktoren können Sie alle Access Modifier verwenden. Manche Klassen sollen nicht von überall instanziierbar sein, und durch eingeschränkte Sichtbarkeit des Konstruktors können Sie das steuern.

Zum Beispiel können Sie eine Klasse nur mit Package-sichtbaren Konstruktoren versehen und so sicherstellen, dass nur Klassen im selben Package – also normalerweise Ihre Klassen – Instanzen erzeugen können. Jedes andere Package kann die Klasse dennoch verwenden, aber Sie kontrollieren das Erzeugen von Objekten.

5.6.2 Übung: Konstruktoren

Implementieren Sie in `Song` drei Konstruktoren:

- einen, der alle drei Feldwerte als Parameter erhält und setzt
- einen, der Titel und Länge als Parameter erhält und für den Interpreten den Default-Wert »Unbekannter Künstler« setzt
- einen, der Titel und Stunden, Minuten und Sekunden der Länge als Parameter erhält

Implementieren Sie keinen Default-Konstruktor. Passen Sie die Klasse `Musicplayer` so an, dass sie einen der Konstruktoren ruft, um neue Songs zu erzeugen. Die Lösung zu dieser Übung finden Sie im Anhang.

Javas lange Klassen

Die Klasse `Song`, die im Wesentlichen drei Felder und eine Methode zur Zeitformatierung enthält, ist inzwischen auf stolze 84 Zeilen angewachsen. Einige davon könnte man natürlich einsparen, dass Sie die Länge auf verschiedene Arten setzen können, ist beispielweise ein nettes Detail, aber alles andere als notwendig.

Das ändert aber nichts daran, dass Java-Klassen, die den hier gezeigten typischen Mustern folgen, also Zugriffsmethoden und mehrere Konstruktoren deklarieren, viel uninteressanten Code enthalten. Das ist ein häufiger Kritikpunkt an der Sprache Java, ein großer Teil des Codes ist dieser sogenannte *Boilerplate Code*.

Bei aller Liebe zur Sprache Java kann ich nur sagen: Das ist nun mal leider so. Natürlich können Sie diesen Code einsparen, indem Sie die Konventionen ignorieren, aber wie im Laufe dieses Kapitels dargelegt, gibt es gute Gründe für diese Konventionen. Als kleiner Trost kann vielleicht gelten, dass Sie diesen Code nicht von Hand schreiben müssen: Jede IDE bietet Ihnen die Möglichkeit, Zugriffsmethoden und Konstruktoren automatisch zu erzeugen. In Netbeans verbirgt sich diese Funktion hinter der Tastenkombination Alt + Einfg.

5.7 Statische Felder und Methoden

Bisher drehte sich alles in diesem Kapitel um Eigenschaften und Methoden, die zu einem Objekt gehören. Das trifft aber nicht auf alle Felder und Methoden zu. In Java muss alles zu einer Klasse gehören, aber nicht unbedingt zu einem Objekt.

Klassen dienen zwar hauptsächlich als Vorlagen für Objekte, aber sie haben auch eine unabhängige Existenz. Klassen können Felder und Methoden haben, die nicht zu einem Objekt gehören. Alle diese Elemente werden mit dem Schlüsselwort `static` gekennzeichnet. Diese Elemente gibt es genau einmal, unabhängig davon, wie viele Instanzen der Klasse erzeugt wurden.

```
public class Song {
    private static long gesamtLaenge;
    public static long getGesamtLaenge(){
        return Song.gesamtLaenge;
    }
    …
}
```

Listing 5.23 Statische Felder und Methoden

Da statische Methoden nicht zu einem Objekt gehören, haben sie auch keinen Zugriff auf nichtstatische Elemente der Klasse.

Wenn statische Elemente von außen sichtbar sind, so kann auf sie mit dem Punktoperator direkt am Klassennamen zugegriffen werden: `Song.getGesamtLaenge()`. Es ist dazu nicht notwendig, ein Objekt zu instanziieren.

Statische Felder zeichnen sich dadurch aus, dass sie nur einen Wert auf Ebene der Klasse haben, unabhängig von deren Instanzen und sogar, wenn niemals eine Instanz erzeugt wird. Sie sind deshalb gut geeignet, Daten zu halten, die nicht zu einem Objekt gehören oder die von allen Instanzen der Klasse gemeinsam genutzt werden. Genau wie auf statische Methoden greifen Sie auch auf statische Felder durch den Punktoperator direkt an der Klasse zu: `Song.gesamtLaenge`.

Aus einem Objekt der Klasse können Sie auch direkt auf statische Felder und Methoden zugreifen, ohne Klassenname und Punkt.

Statische Initialisierung

Es gibt neben statischen Methoden und Feldern noch ein weiteres statisches Konstrukt, das allerdings viel seltener genutzt wird: den *Static Initializer*. Er ist für die Klasse ungefähr das, was der Konstruktor für die Instanz ist: er führt notwendige Initialisierungsarbeiten aus.

```
public class StatischInitialisiert {
    static {
        //hier steht Initialisierungscode
    }
}
```

Ein Programm braucht normalerweise keine statische Initialisierung, um statische Felder mit einem Startwert zu versehen, genau wie bei jeder anderen Variablen können Sie den bei der Deklaration angeben. Aber wenn der Startwert beispielsweise aus einer Datei gelesen werden soll, dann kann das in der Initialisierung passieren. Besonders wichtig ist das bei Konstanten (siehe Abschnitt 5.8, »Unveränderliche Werte«), denn diese können im Static Initializer noch gesetzt werden, danach ist ihr Wert für immer unveränderlich.

Häufig werden Static Initializer aber aus einem anderen Grund verwendet: Die Laufzeitumgebung garantiert, dass sie genau einmal ausgeführt werden, nämlich genau dann, wenn die Klasse geladen wird. Wenn also ein bestimmtes Stück Code nur einmal ausgeführt werden darf und auf gar keinen Fall öfter, dann wird das meistens auf diese Art sichergestellt.

5.7.1 Übung: Statische Felder und Methoden

Erweitern Sie die Klasse `Song` um das oben gezeigte Feld `gesamtLaenge` und die zugehörige Zugriffsmethode `getGesamtLaenge`. Das Feld soll immer die Länge aller erzeugten

Songs enthalten. Achten Sie vor allem darauf, diesen Wert anzupassen, wenn die Länge eines Songs nachträglich geändert wird. Die Lösung zu dieser Übung finden Sie im Anhang.

5.7.2 Die »main«-Methode

Mit diesem neuen Wissen über statische Methoden wird jetzt auch die Deklaration der `main`-Methode klarer. Es handelt sich um eine einfache statische Methode, die nur dadurch besonders ist, dass die Laufzeitumgebung sie aufruft, wenn sie mit der Klasse als Hauptklasse gestartet wird.

Die `main`-Methode wird nicht nur an ihrem Namen erkannt, ihre gesamte Signatur *muss* so lauten, wie Sie sie bisher auch deklariert haben:

```
public static void main (String[] args)
```

Lediglich den Namen des Parameters dürfen Sie beliebig ändern. Der Name `args` hat allerdings Tradition, und es gibt selten einen Grund, ihn zu ändern.

5.7.3 Statische Importe

Sie können auf statische Felder und Methoden auch aus einer anderen Klasse zugreifen, ohne den Punktoperator zu verwenden, indem Sie die Elemente *statisch importieren*.

Genau wie das `import`-Statement eine Klasse aus einem Package in Ihren Code importiert, importiert `import static` statische Member aus einer anderen Klasse in Ihre Klasse. Diese statischen Importe geben Sie an derselben Stelle an, an der Sie auch andere Importe machen, und führen nach dem Klassennamen noch die statischen Member auf, die Sie aus dieser Klasse importieren möchten:

```
import static de.kaiguenster.javaintro.music.Song. getGesamtLaenge;
```

Oder auch alle statischen Elemente:

```
import static de.kaiguenster.javaintro.music.Song.*;
```

Auf die importierten Member können Sie dann so zugreifen, als seien sie in dieser Klasse deklariert.

```
import static de.kaiguenster.javaintro.music.Song.*;
public class Musicplayer {
    …
    public void druckeGesamtLaenge(){
```

```
        System.out.println(getGesamtLaenge());
    }
}
```

Listing 5.24 Eine statisch importierte Methode verwenden

Am nützlichsten sind statische Importe für Konstanten (siehe Abschnitt 5.8.2), die Sie so verwenden können, ohne immer wieder die Klasse anzugeben, aus der die Konstante stammt. Auch für allgemein nützliche Hilfsmethoden, die Sie an vielen Stellen im Programm benutzen, sind statische Importe hilfreich, denn nachdem eine solche Methode einmal importiert ist, müssen Sie nicht immer wieder die Ursprungsklasse dazuschreiben oder sich auch nur daran erinnern, wo die Methode wirklich steht.

5.8 Unveränderliche Werte

Neben den veränderlichen Variablen und Feldern, die Sie bisher kennengelernt haben, gibt es in Java auch die Möglichkeit, sie permanent auf einen Wert festzuschreiben. Dazu müssen Sie der Deklaration nur das Schlüsselwort `final` voranstellen.

```
final String titel = "The Final Countdown";
```

Der Compiler lässt dann nicht zu, dass der Variablen ein neuer Wert zugewiesen werden kann. Sie müssen einer solchen Variablen ihren Wert bei der Deklaration geben, ein späterer Schreibzugriff führt zu einem Compiler-Fehler.

Sie können lokale Variablen und Parameter `final` deklarieren, das bewahrt Sie davor, ihnen versehentlich einen neuen Wert zuzuweisen, aber der Hauptnutzen davon ist, dass Sie auf diese Variablen in inneren Klassen zugreifen können (siehe dazu Abschnitt 6.4, »Innere Klassen«).

Auch jetzt können Sie aber schon einen Nutzen aus unveränderlichen statischen und nichtstatischen Feldern ziehen.

Unveränderlich heißt nicht unveränderlich

`final` bedeutet, dass einer Variablen kein neuer Wert zugewiesen werden kann. Für Objektvariablen bedeutet das, dass sie niemals ein anderes Objekt referenzieren können. Wenn das referenzierte Objekt aber selbst einen veränderlichen Zustand hat, also Felder, deren Wert Sie ändern können, dann ändert `final` nichts daran.

Das Gleiche gilt auch für Arrays: Einer als `final` deklarierten Array-Variablen können Sie kein anderes Array mehr zuweisen, aber die Werte im Array können Sie beliebig verändern.

5.8.1 Unveränderliche Felder

Felder mit dem Modifier `final` können nicht nur bei der Deklaration ihren Wert erhalten, sondern auch noch im Konstruktor, aber nicht später. Dadurch ist es möglich, im Konstruktor übergebene Parameter in ihnen zu speichern und gleichzeitig sicherzustellen, dass sich ihr Wert danach nicht mehr verändert.

```
public class Song {
    private final String titel;
    private final String interpret;
    private final int laengeInSekunden;
    public Song(String titel, String interpret, int laengeInSekunden) {
        //die Parameter hier validieren
        this.titel = titel;
        this.interpret = interpret;
        this.laengeInSekunden = laengeInSekunden;
    }
}
```

Listing 5.25 Der unveränderliche »Song«

Am Beispiel der `Song`-Klasse bedeutet das, dass Titel, Interpret und Länge nie mehr geändert werden können. Ein Song hat so eine unveränderliche Identität und kann zum Beispiel nicht nachträglich umbenannt werden. Sollten Sie später feststellen, dass sich ein Tippfehler eingeschlichen hat, dann können Sie nur ein neues Objekt mit den geänderten Werten erzeugen.

Unveränderliche Objekte (»Immutable Objects«)

Auf den ersten Blick findet sich kein großer Vorteil darin, Felder unveränderlich zu machen. Erst mit etwas vertieftem Wissen treten einige Vorteile hervor.

Sie müssen sich zum Beispiel bei Objekten, deren Felder **alle** `final` sind, keine Sorgen machen, dass sie durch eine andere Referenz geändert werden. Ein solches *Immutable Object* kann einfach nicht mehr verändert werden. In diese Kategorie fallen auch Strings und die Wrapper-Klassen der primitiven Typen: Sie sind unveränderlich. Methoden, die so aussehen, als würden sie den Wert verändern, geben stattdessen ein neues Objekt zurück.

5.8.2 Konstanten

Felder, die sowohl `static` als auch `final` sind, heißen *Konstanten*. Sie haben für die gesamte Laufzeit eines Programms einen gleichbleibenden Wert, auf den entweder alle Objekte der Klasse zugreifen können oder sogar das ganze Programm, je nach Access Modifier.

Konstanten kommen dann zum Einsatz, wenn ein Wert im Code immer wieder und an mehreren Stellen benötigt wird, und tragen viel dazu bei, Code lesbar zu machen. Denken Sie zum Beispiel an den BMI-Rechner aus Abschnitt 3.1.3 zurück, in dem der berechnete BMI mit mehreren Zahlen verglichen wurde, die für sich genommen vollkommen willkürlich scheinen. Hier folgt der gleiche Code, aber mit Konstanten:

```
public class BMIRechner {
    public static final double OBERGRENZE_UNTERGEWICHT = 18.5;
    public static final double OBERGRENZE_NORMALGEWICHT = 25;
    public static final double OBERGRENZE_UEBERGEWICHT = 30;

    public static void main(String[] args) throws IOException {
        …
        if (bmi < OBERGRENZE_UNTERGEWICHT){
            System.out.println("Damit haben Sie Untergewicht");
        } else if (bmi < OBERGRENZE_NORMALGEWICHT){
            System.out.println("Damit haben Sie Normalgewicht");
        } else if (bmi < OBERGRENZE_UEBERGEWICHT){
            System.out.println("Damit haben Sie Übergewicht");
        } else {
            System.out.println("Damit haben Sie schweres Übergewicht");
        }
    }
}
```

Listing 5.26 BMI-Rechner mit Konstanten

Durch diese kleine Änderung sieht der Code viel lesbarer aus, es ist auf einen Blick klar, warum die Vergleiche so gemacht werden, wie sie gemacht werden. Und dadurch, dass die Konstanten `public` sind, können Sie von überall auf sie zugreifen. Wäre der `BMIRechner` ein Teil eines größeren Programms, dann könnten Sie überall die Konstanten nutzen, anstatt immer wieder die gleichen Zahlen in den Code zu schreiben. Der Vorteil davon wird klar, wenn Sie sich vorstellen, dass diese Werte geändert würden und zum Beispiel Übergewicht erst bei einem BMI von 27 begönne statt wie bisher bei 25. Ohne die Konstante müssten Sie alle Vorkommen der Zahl 25 suchen und bei jedem einzelnen prüfen, ob es sich hier um einen BMI-Wert handelt oder nicht, und wenn ja den Wert ändern. Mit der Konstante ändern Sie nur einmal den Konstantenwert, und es wird mit einfachem Neukompilieren überall der neue Wert benutzt.

Im Code sehen Sie auch die übliche Namenskonvention für Konstanten: Sie werden in Großbuchstaben und mit Unterstrichen anstelle von Leerzeichen benannt.

5.9 Spezielle Objektmethoden

Es gibt drei wichtige Methoden, die alle Ihre Klassen von `Object` erben, die Sie aber häufig überschreiben, also in Ihrer Klasse selbst neu implementieren werden.

Eine dieser Methoden haben Sie in `Song` bereits implementiert: `toString`, die Methode, die zur Ausgabe Ihrer Objekte als Zeichenkette gerufen wird.

Eine weitere Methode haben Sie noch nicht implementiert, aber zumindest schon genutzt: `equals` oder, um genau zu sein, `public boolean equals(Object other)`. Die Implementierung von `equals` in `Object` liefert genau dann `true` zurück, wenn `this` und `other` dasselbe Objekt referenzieren, sie verhält sich also genauso wie der `==`-Operator.

Für Ihre eigenen Klassen ist das aber häufig nicht das gewünschte Verhalten. Wenn zwei Songs den gleichen Titel haben, vom gleichen Interpreten stammen und gleich lang sind, dann sollte es sich auch um den gleichen Song handeln. Um genau diese Definition von Gleichheit für Java-Objekte zu verwirklichen würden Sie die `equals`-Methode überschreiben. Es gibt drei Grundregeln für die `equals`-Methode:

- Ein Objekt ist immer gleich sich selbst. Wenn `this == other` wahr ist, dann muss auch `this.equals(other)` wahr sein.
- Ein Objekt ist niemals gleich `null`. `anObject.equals(null)` muss `false` zurückgeben, aber keine `NullPointerException` werfen.
- Ansonsten sind zwei Objekte dann gleich, wenn sie Ihre Definition von Gleichheit erfüllen. Das kann bedeuten, dass alle Feldwerte identisch sind, muss es aber nicht. Im Fall von `Song` könnten Sie zum Beispiel festlegen, dass nur Titel und Interpret verglichen werden, weil die Länge aus rein technischen Gründen abweichen kann, ohne dass es deswegen ein anderes Stück wäre. Ich empfehle alle Felder, die in `equals` verglichen werden, auch `final` zu deklarieren. So ist sichergestellt, dass zwei Objekte, die einmal gleich waren, auch gleich bleiben.

Ein viertes Kriterium, nämlich ob zwei Objekte Instanzen derselben Klasse sein müssen, um gleich zu sein, ist etwas strittiger. Manchmal ist es sinnvoll, dass auch zwei Objekte, deren Klassen lediglich eine gemeinsame Oberklasse haben, gleich sein können.

Für die Klasse `Song` kann die `equals`-Methode zum Beispiel so aussehen:

```
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
```

```
    final Song other = (Song) obj;
    if (!Objects.equals(this.titel, other.titel)) {
        return false;
    }
    if (!Objects.equals(this.interpret, other.interpret)) {
        return false;
    }
    return true;
}
```

Listing 5.27 Die »equals«-Methode der »Song«-Klasse

Diese `equals`-Methode ist sehr typisch, und Sie können an ihr die oben genannten Kriterien nachvollziehen:

- Zuerst wird geprüft, ob das andere Objekt `null` ist. Falls ja, wird `false` zurückgegeben.
- Als Nächstes wird geprüft, ob beide Objekte Instanzen derselben Klasse sind. Die Methode `getClass`, die an jedem Objekt zur Verfügung steht, gibt die Klasse des Objekts zurück, und da Klassen garantiert nur einmal vorhanden sind, können sie getrost mit `!=` verglichen werden.
- Wenn diese beiden Bedingungen erfüllt sind, werden die Felder verglichen. Dazu wird die statische Methode `Objects.equals` verwendet. Sie verwendet die `equals`-Methoden der Feldwerte, prüft aber vorher, ob diese `null` sind. So kann keine `NullPointerException` auftreten, und Sie müssen die `null`-Prüfung nicht immer wieder selbst schreiben.
- Es wird nicht explizit geprüft, ob das übergebene Objekt `obj` mit `this` identisch ist. Diese Prüfung ist nicht notwendig, denn auch die gezeigte Methode gibt in dem Fall `true` zurück. Häufig sieht man dennoch am Anfang der `equals`-Methode die Phrase `if (this == obj) return true`, da so die weiteren Prüfungen übersprungen werden und die Methode schneller endet.

Die dritte Methode, die Sie häufig überschreiben werden, ist `hashCode`. Diese Methode hat einen etwas merkwürdigen Status, weil Sie fachlich eigentlich keinen Grund haben, sie zu überschreiben. Sie berechnet einen Zahlenwert zu dem Objekt, der »einigermaßen eindeutig« sein sollte und der benutzt wird, wenn das Objekt als Schlüssel in einer `HashMap` verwendet wird. Dazu mehr in Kapitel 10, »Arrays und Collections«.

Der einzige Grund, diese Methode hier überhaupt schon zu erwähnen, ist der, dass Sie sie immer – immer! – zusammen mit `equals` überschreiben müssen. Damit einige Komponenten der Klassenbibliothek funktionieren, vor allem die schon angespro-

chene `HashMap`, ist es unbedingt notwendig, dass zwei Objekte, die laut `equals` gleich sind, auch den gleichen Hash-Code haben.

Sie sollten sich aber, um ganz ehrlich zu sein, nicht allzu viele Gedanken über `equals` und `hashCode` machen. Es sind zwei sehr wichtige Methoden, die Sie unbedingt kennen müssen, aber Sie werden sie nur selten selbst schreiben: Jede moderne IDE bietet eine Funktion, um diese Methoden generieren zu lassen. In NetBeans finden Sie diese Funktion neben den Zugriffsmethoden unter Alt + Einfg.

5.10 Zusammenfassung

Sie haben in diesem Kapitel gelernt, wie Sie eigene Klassen definieren und Instanzen dieser Klassen erzeugen. Sie haben gelernt, einer Klasse Felder und Methoden hinzuzufügen, was der Unterschied zwischen statischen und nichtstatischen Members ist und wie Sie ein Feld unveränderbar machen. Zur objektorientierten Programmierung fehlt Ihnen jetzt noch die Vererbung.

Das nächste Kapitel vertieft Ihr Wissen über Objekte. Sie wissen nun bereits, wie Sie Klassen und Objekte anlegen, aber damit sind Ihre Programme noch nicht vollständig objektorientiert. Sie werden lernen, wie Klassen voneinander erben können und welche Vorteile Ihnen das bringt.