Check und Prepare

Nachschlagewerk - Fortgeschritten-Level in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Autoren: Nhan Huynh und Darya Nikitina
Fachbereich: Informatik

Version: 02. Oktober 2021
Semesterübergreifend

Vorwort

Dieses Nachschlagewerk wurde von uns erstellt, um alle relevanten Themen zu wiederholen, die die Grundlagen von Java und generell objektorientierten Sprachen darstellen. Dabei sollen die wichtigsten Punkte dieser Themen kurz und prägnant erklärt und mit Beispielen weiter veranschaulicht werden.

Wir beginnen im Abschnitt 1.1 damit an, was eine Klasse und ein Objekt genau ist. Dann erklären wir in den darauf folgenden Abschnitten 1.1.2 und 1.1.3 nacheinander im Detail die Arten und den Sinn von Attributen und Methoden. Als Nächstes kommen wir in 1.2 zum Thema Vererbung: Wir erklären in 1.2.1 und 1.2.3, was Subtypen sind und wie man Variablen „casten“ kann. Danach wird im Abschnitt 1.3 auf den Unterschied zwischen Interfaces und abstrakten Klassen eingangen. Als darauffolgendes Thema erklären wir in 1.4, was unter dem Begriff „Scope“ verstanden wird. Schließlich gehen wir in 1.5 auf den Unterschied zwischen primitiven Datentypen und Referenztypen ein.

Achtung: Dieses Nachschlagewerk hat den Sinn alle relevanten Themen zum Einstieg in die objektorientierte Programmierung zu wiederholen. Es dient nicht dazu Sie in das Thema Informatik einzuführen. Wenn Sie also nicht mit den Konzepten, wie beispielsweise Schleifen, `if`-Abfragen oder Variablen, vertraut sind, sollten Sie sich zuerst einen Überblick davon verschaffen. (Dabei kann der Anfänger-Level des Kurses Check+Prepare eine Hilfe sein[a].)

Vor allem: Wir garantieren auch nicht, dass alle genannten und behandelten Themen vollständig und im Detail den Anforderungen des Kurses *Funktionale und Objektorientierte Programmierkonzepte* entsprechen. Nutzen Sie dieses Nachschlagewerk eher als zusätzliches Material oder als Quelle für die Vorbereitung zum Kurs. Keinesfalls aber als einzige Quelle. Wenn Sie Programmieren als Hobby oder zusätzliche Leistung außerhalb des Studiums erlernen wollen, betrifft Sie diese Warnung natürlich nicht.

[a] Link zum Kurs: `https://moodle.informatik.tu-darmstadt.de/course/view.php?id=1028`

Inhaltsverzeichnis

1 Grundbegriffe **4**
1.1 Klassen, Objekte und Instanz 4
1.1.1 Zugriffsmodifikatoren 6
1.1.2 Attribute 8
1.1.3 Methoden 12
1.1.4 Konstruktor 15
1.2 Vererbung 17
1.2.1 Subtypen 18
1.2.2 Überschreiben von Methoden 20
1.2.3 Methodentabelle 21
1.3 Interfaces und abstrakte Klassen 23
1.3.1 Interface 23
1.3.2 Abstrakte Klassen 25
1.4 Scope 26
1.4.1 Attribute vs Lokale Variablen und Konstanten 26
1.4.2 Methoden 27
1.5 Primitive Datentypen und Referenztypen 28
1.5.1 Wertgleichheit und Objektidentität 28
1.5.2 Primitive Datentypen 28
1.5.3 Referenztypen 28
1.5.4 Vergleich zwischen Referenztypen und primitiven Datentypen 29
1.5.5 Wertgleichheit vs. Objektgleichheit 30
1.6 Arrays 32
1.7 Enums 36

Stichwortverzeichnis **37**

1 Grundbegriffe

1.1 Klassen, Objekte und Instanz

Java ist eine objektorientierte Programmiersprache und deshalb ist der Begriff der Klasse ganz zentral. Im Allgemeinen ist eine Klasse eine Beschreibung eines Objekts mit seinen Attributen und Methoden. Sie dient als Vorlage, aus der dann beliebig viele Objekte erzeugt werden können.

Man kann sich eine Klasse als eine Art „Bauanleitung“ für ein Objekt vorstellen. Die definierten Attribute und Methoden helfen dabei das Objekt zu charakterisieren. Dabei kann man sich Attribute als Merkmale oder Eigenschaften und die Methoden als Verhalten des Objektes veranschaulichen (genaueres dazu in den Abschnitten 1.1.2 und 1.1.3 erläutert). Das Objekt selbst hingegen stellt die Instanziierung einer Klasse mit spezifischen Werten für die Attribute einer Klasse dar.

Um sich das Ganze etwas besser zu veranschaulichen, nehmen wir das folgende Beispiel.

Ein Bauplan für eine sehr einfache Modellierung eines Autos kann folgendermaßen aussehen:

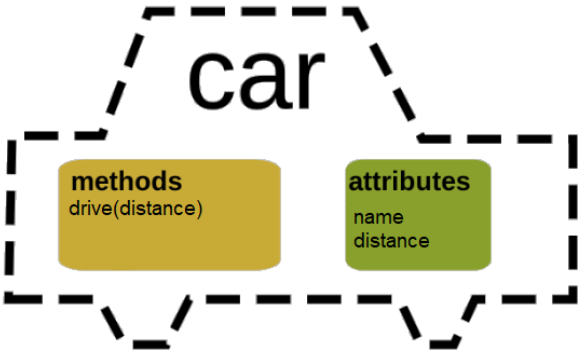

Abbildung 1.1: Bauplan eines Autos - Klasse Car

Wir definieren dazu die folgende Klasse:

```
1  class Car {
2
3    String name;
4    double distance;
5
6    Car(String name) {
7      this.name = name;
8      distance = 0;
9    }
10
11
12   void drive(double distance) {
13     this.distance += distance;
14   }
15 }
```

Abbildung 1.2: Bauplan eines Autos in Java - Klasse Car

Wir können den Bauplan, den wir in der Abbildung 1.1 gesehen haben in dem Java-Code erkennen. Ein Auto hat einen Namen und die gefahrene Distanz als Attribut. Dabei wird der Name als Zeichenkette, d.h. `String`, abgespeichert und die gefahrene Distanz als Gleitkommazahl, d.h. `double`. Des Weiteren kann ein Auto eine Strecke von einer vorgegebenen Distanz mit der Methode `drive` fahren. Mithilfe von der Klasse `Car` können wir nun verschiedene Instanzen davon erstellen.

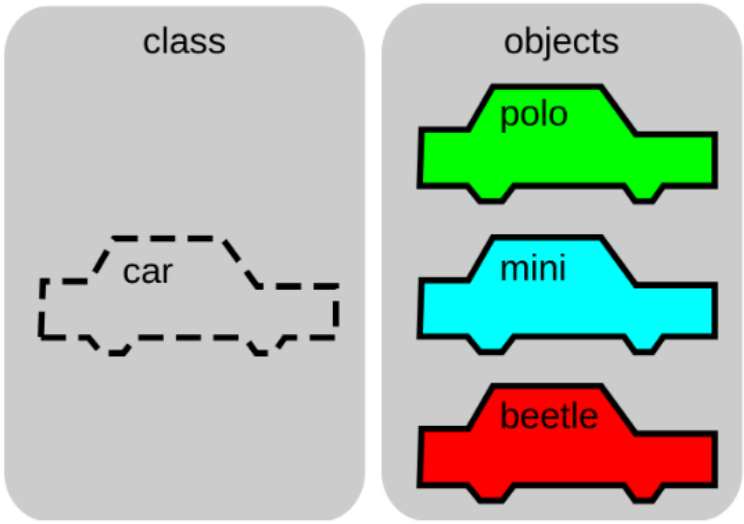

Abbildung 1.3: Verschiedene Instanzen der Klasse Car

Information: Wir haben hier auch zum ersten Mal den Konstruktor einer Klasse kennen gelernt (mehr dazu bei 1.1.4). Wenn ein Objekt einer Klasse erstellt wird, wird zuallererst der Konstruktor der Klasse aufgerufen. Bei diesem Beispiel ist es dadurch möglich dem Auto einen Namen zu geben, der im Attribut `name` gespeichert wird. Man kann Objekte einer Klasse mit dem Schlüsselwort `new` instanziieren. Dann kann man auf die Attribute und Methoden des Objekts zugreifen.

1.1.1 Zugriffsmodifikatoren

Bevor wir zu den Definitionen von „Attributen“ und „Methoden“ übergehen, müssen wir zuvor noch einen wichtigen Begriff einführen - „Zugriffsmodifikatoren“ (oder Modifier). Zugriffsmodifikatoren bestimmen für eine gegebene Klasse, ob andere Klassen ein bestimmtes Attribut verwenden oder bestimmte Methoden aufrufen können. Konkret können Zugriffsmodifikatoren also die Sichtbarkeit von Klassen, Attributen oder Methoden einschränken. Dabei können zwei Ebenen unterschieden werden und je nachdem um welche Ebene es sich handelt, dürfen nur bestimmte Zugriffsmodifikatoren verwendet werden:

- High-Level:
  - `public`
  - Package-privat (kein expliziter Modifikator)
- Member-Level:
  - `public`
  - `protected`
  - Package-privat (kein expliziter Modifikator)
  - `private`

Mit High-Level sind Klassen, Interfaces oder Enums gemeint, die dem Dateinamen entsprechen. Alles Andere, wie beispielsweise Attribute oder innere Klassen gehören zum Member-Level.
In der Tabelle 1.1 können Sie die Sichtbarkeit bezogen auf die jeweiligen Modifikatoren herauslesen.

Modifikator	Klasse	Package	Subklassen	Welt
`public`	Ja	Ja	Ja	Ja
`protected`	Ja	Ja	Ja	Nein
Kein Modifikator[1]	Ja	Nein	Nein	Nein
`private`	Ja	Nein	Nein	Nein

Tabelle 1.1: Zugriffsebenen

- Klasse: Diese Spalte gibt an, ob die Klasse selbst Zugriff auf ein Element mit dem entsprechenenden Modifikator hat. Wir können sehen, dass eine Klasse unabhängig vom Modifikator immer Zugriff auf ihre eigenen Elemente hat.
- Package: Diese Spalte gibt an, ob Klassen im selben Package Zugriff auf ein Element mit dem entsprechenenden Modifikator haben.
- Subklassen: Diese Spalte gibt an, ob Subklassen (Unterklassen), die außerhalb des Package definiert wurden, Zugriff auf ein Element mit dem entsprechenenden Modifikator haben.
- Welt: Diese Spalte gibt an, ob alle Klassen, unabhängig von ihrem Speicherort, Zugriff auf ein Element mit dem entsprechenenden Modifikator haben.

Information: „Ein Package ist ein Namespace mithilfe dessen eine Reihe verwandter Klassen und Interfaces organsisiert werden können.“ (`https://docs.oracle.com/javase/tutorial/java/concepts/package.html`)

Ein Beispiel für ein Package ist `java.awt`. Dieses Package enthält Klassen zum Erstellen von Benutzeroberflächen und zum Zeichnen von Grafiken und Bildern.

Mithilfe des folgenden Verweises können Sie sich einen Überblick über Packages verschaffen:

`https://docs.oracle.com/en/java/javase/11/docs/api/allpackages-index.html`

Genaueres zu Subklassen wird im Abschnitt 1.2 und Referenzen im Abschnitt 1.5 erläutert.

[1] wird auch als package-private bezeichnet

Um zu verdeutlichen, wie genau Zugriffsmodifikatoren funktionieren, nehmen wir den Codeabschnitt der Abbildung 1.2 als Beispiel. Wie wir bereits gesehen haben, darf auf dem High-Level der Zugriffsmodifikator nur package-private oder `public` sein. Unsere Klasse soll von überall her ansprechbar sein, weshalb wir sie auf `public` setzen. Als nächstes betrachten wir die Attribute und Methoden der Klasse, sowie ihren Konstruktor, die alle dem Member-Level entsprechen. Die Sichtbarkeit der Attribute der Klasse `Car` stellen wir auf `private`, damit niemand sie modifizieren kann. Die Methode `drive(distance)` und der Konstruktor hingegen, sollen von außen ansprechbar sein, weshalb sie auf `public` gesetzt werden. Als Resultat erhalten wir den folgenden Codeabschnitt:

```
1  public class Car {
2
3    private String name;
4    double mileage;
5
6    public Car(String name) {
7      this.name = name;
8      mileage = 0;
9    }
10
11
12   public void drive(double distance) {
13     mileage += distance;
14   }
15 }
```

Abbildung 1.4: Klasse Car mit Zugriffsmodifikatoren

1.1.2 Attribute

Attribute sind Merkmale oder Eigenschaften eines Objektes einer Klasse. Sie werden direkt in der Klasse definiert, also nicht innerhalb von Methoden. Per Konvention definiert man sie unmittelbar unter der Definition der Klasse.

> **Information:** Diese und weitere Code-Konventionen können Sie unter dem folgenden Verweis finden:
>
> Abschnitt 3.1.3 Class and Interface Declarations
>
> `https://www.oracle.com/java/technologies/javase/codeconventions-fileorganization.html`

Durch verschiedene Schlüsselwörter lassen sich Attribute spezifizieren. Im Allgemeinen können sie nach folgendem Schema definiert werden:

$$\text{Zugriffsmodifikator}^{*}\ \texttt{static}^{*}\ \text{Datentyp Bezeichner} = \text{Wert};$$

Die Begriffe, die mit einem * (Asterisk) markiert sind, sind optional.

In den folgenden Abschnitten werden wir uns anschauen, was mit den einzelnen Begriffen in diesem Schema gemeint ist.

Zugriffsmodifikatoren

Zugriffsmodifikatoren ermöglichen es die Sichtbarkeit von Attributen innerhalb von Teilen des Programms zu regulieren. Leichter gesagt, kann man nicht auf manche Attribute zugreifen, wenn bestimmte Zugriffsmodifikatoren verwendet werden. Das kann beispielsweise das Programm schützen, da es dem Nutzer nicht ermöglicht wahllos auf Attribute zuzugreifen und deren Werte zu verändern.

Es gibt vier Typen von Attributen, die sich zwei Oberbegriffen zuordnen lassen:

- Klassenattribute
- Objektattribute

Im folgenden wollen wir mithilfe von zwei Codeabschnitten die Verwendung von den zwei Typen von Attributen veranschaulichen.

Die Klasse `Circle` modelliert einen Kreis. Als Attribut hat der Kreis einen Radius und mittels der Methoden können wir den Radius des Kreises abfragen und den Umfang und Flächeninhalt des Kreises bestimmen.

```
1  public class Circle {
2
3    private static final double PI = 3.14159265358979323846;
4    private final double radius;
5
6    public Circle(double radius) {
7      this.radius = radius;
8    }
9
10   public double getRadius() {
11     return radius;
12   }
13
14   public double getCircumference() {
15     return Circle.PI * 2 * radius;
16   }
17
18   public double getArea() {
19     return Circle.PI * radius * radius;
20   }
21 }
```

Abbildung 1.5: Klasse Circle

Die Klasse `Student` modelliert eine/n Studenten/in. Als Attribut hat Student eine Identifikationsnummer und einen Namen. Mittels der Methoden können wir die ID und den Namen des Student auslesen und den Namen des/r Student/in modifizieren.

```
1  public class Student {
2
3    private static long ID = 0;
4
5    public final long id;
6    private String name;
7
8    public Student(String name) {
9      this.id = ID++;
10     this.name = name;
11   }
12
13   public long getID() {
14     return id;
15   }
16
17   public String getName() {
18     return name;
19   }
20
21   public void setName(String name) {
22     this.name = name;
23   }
24 }
```

Abbildung 1.6: Klasse Student

Klassenattribute
Klassenattribute bzw. statische Attribute sind Eigenschaften, die nicht einzelnen Objekten, sondern deren gesamter Klasse zugeordnet werden. Sie werden mit dem Schlüsselwort `static` gekennzeichnet. Jede Instanz der Klasse enthält die gleichen Klassenattribute d.h. alle Objekte der Klasse teilen sich die Klassenattribute.

Auf Klassenattribute kann folgendermaßen zugegriffen werden:

Klassenname.Klassenattribut

In der Abbildung 1.5 sehen wir jeweils den Zugriff auf das Klassenattribut `PI`. Im Allgemeinen kann der Zugriff auf Klassenattribute in der eigenen Klasse oder bei Subklassen auch ohne den Klassennamen erfolgen (siehe dazu Abbildung 1.6 - `id` ; Achtung: Hierbei ist `id` und nicht `id` gemeint!).

Außerdem gibt es noch Klassenkonstanten, die mit dem Schlüsselwort `final` gekennzeichnet werden. Wie der Name Konstante schon sagt, muss der Wert einer Konstanten konstant bleiben, d.h. Konstanten müssen initialisiert werden und eine erneute Zuweisung des gleichen oder eines anderen Wertes ist nicht möglich. Klassenkonstanten können deklarieren und anschließend in einem Static-Block initialisiert werden und müssen deshalb nicht sofort bei der Deklaration initialisiert werden.

```
1 private static final double PI;
2
3 static {
4   PI = 3.14159265358979323846;
5 }
```

Abbildung 1.7: Initialisierung von Klassenkonstanten mittels eines Static-Blocks

Information: Unter Zuweisung versteht man, dass in einer Variablen ein Wert gespeichert wird. Dabei wird der Name einer Variablen gefolgt von einem = und dann dem Wert, den wir da rein speichern wollen, aufgeschrieben. Natürlich muss der zu speichernde Wert dem Datentyp entsprechen, den die Variable hat. Als Beispiel, wenn wir eine `int`-Variable namens `i` haben und dort den Wert 15 speichern wollen:

```
i = 15;
```

Unter Deklarierung einer Variablen versteht man die Erstellung einer Variablen. Dabei wird der Datentyp der Variable gefolgt vom Variablennamen aufgeschrieben. Angenommen, wir wollen eine `int`-Variable namens `i` deklarieren, würden wir den folgenden Code schreiben:

```
int i;
```

Unter Deklarierung und Initialisierung einer Variablen versteht man, dass die Variable sofort bei der Deklarierung einen Wert zugewiesen bekommen. Dabei wird der Datentyp der Variablen, gefolgt vom Variablennamen und einem = sowie dem Wert, der gespeichert werden soll und zum Datentyp passt, aufgeschrieben. Als Beispiel wollen wir die `int`-Variable namens `i` mit dem Wert 15 initialisieren:

```
int i = 15;
```

Objektattribute

Objektattribute enthalten das Schlüsselwort `static` nicht und gelten unabhängig voneinander für jede Instanz, d.h. dass jede Instanz ihre eigenen Objektattribute mit eigenen Werten hat. Objektattribute können nur durch ein Objekt angesprochen werden:

Objekt.Objektattribut

In der Abbildung 1.6 wären also folgende Attribute Objektattribute:

- `private final long id`: Die Identifikationsnummer des Student.
- `private String name`: Der Name des Student.

Jede Instanz eines `Student` hat seine eigene `id` und `name`.

Betrachten wir den folgenden Codeabschnitt:

```
1 Student s1 = new Student("Max");
2 Student s2 = new Student("Erika");
```

Abbildung 1.8: Beispielinstanzen der Klasse Student

Das Objekt `s1` hat somit seine eigene `id` (= 0) und `name` (= Max) und `s2` hingegen `id` (= 1) und `name` (= Erika).

Es gibt analog zu Klassenkonstanten auch Objektkonstanten, die mit dem Schlüsselwort `final` gekennzeichnet werden. Objektkonstanten müssen ebenfalls initialisiert werden und eine erneute Zuweisung mit demselben oder einem anderen Wert ist nicht möglich. Es gibt hier auch die Möglichkeit Objektkonstanten nur zu deklarieren und anschließend in einem Konstruktor zu initialisieren statt sie sofort bei der Deklaration zu initialisieren.

Standardwerte

Attribute (nicht Konstanten!) denen kein expliziter Wert zugewiesen wurde, nehmen automatisch abhängig von ihrem Datentyp einen bestimmten Standardwert an.

Datentyp	Standardwert
`boolean`	`false`
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`
`char`	`'\u0000`
Referenztypen	`null`

Tabelle 1.2: Standardwerte für Attribute

Achtung: Die Standardwerte gelten nur für Attribute! Wenn Sie lokale Variablen innerhalb von Methoden erstellen, müssen Sie darauf achten den Variablen immer Werte zuzuweisen. Es wird zu einem Kompilierfehler führen, wenn Sie versuchen lokale Variablen zu verwenden, die nur deklariert und nicht initialisiert wurden.

Mehr Informationen zu lokalen Variablen finden Sie im Abschnitt 1.4.1.

1.1.3 Methoden

Methoden definieren das Verhalten eines Objektes, d.h. sie entsprechen Anweisungen. Sie werden direkt in der Klasse, typischerweise unmittelbar unter der Definition von Konstruktoren, definiert (Konvention, siehe Abschnitt 1.1.2).

Durch verschiedene Schlüsselwörter lassen sich Methoden spezifizieren. Im Allgemeinen können sie nach folgendem Schema definiert werden:

Zugriffsmodifikator* `static`* Rückgabetyp Bezeichner(Parameter*);

Die Begriffe, die mit einem * (Asterisk) markiert sind, sind optional und Parameter werden auch als Argumente bezeichnet.

In den folgenden Abschnitten werden wir uns anschauen, was mit den einzelnen Begriffen in diesem Schema gemeint ist.

Falls die Methode nichts zurück liefern soll, wird das Schlüsselwort `void` als Rückgabetyp verwendet. Eine Methode kann n-beliebige Parameter haben, wobei $n \in \mathbb{N}_0$[2].

Es gibt zwei Arten von Methoden:

- Klassenmethoden
- Objektmethoden

Klassenmethoden

Klassenmethoden bzw. statische Methoden sind Verhalten, die nicht einzelnen Objekten, sondern deren gesamter Klasse zugeordnet werden. Sie werden mit dem Schlüsselwort `static` gekennzeichnet und können weder auf Objektattribute zugreifen noch Objektmethoden aufrufen, sondern nur Klassenattribute und Klassenmethoden verwenden.

Auf Klassenattribute kann folgendermaßen zugegriffen werden:

Klassenname.Klassenmethode

Als Beispiel nehmen wir wieder den Codeausschnitt aus der Abbildung 1.5. Wir erweitern die Klasse um eine Methode, die mehrere Kreise übergeben bekommt und die Summe der Umfänge aller Kreise zurück gibt. Da diese Methode kein Verhalten eines einzelnen Objektes darstellen soll, definieren wir sie als Klassenmethode.

```
1 public static double getCircumferences(Circle[] circles) {
2   double circumferences = 0;
3   for (Circle circle : circles) {
4     circumferences += circle.getCircumference();
5   }
6   return circumference;
7 }
```

Abbildung 1.9: Beispiel einer Klassenmethode - Klasse Circle

Wir können nun die Klassenmethode folgendermaßen aufrufen:

```
1 Circle[] circles = {new Circle(2.5), new Circle(6.6)};
2 double circumference = Circle.getCircumferences(circles);
```

Abbildung 1.10: Beispielaufruf Klassenmethode - Klasse Circle

Die Summe der Umfänge der Kreise in der Abbildung 1.10 beträgt 57.17698629533423.

> **Information:** Das Codebeispiel aus der Abbildung 1.9 verwendet als Parameter Arrays. Genaueres zu Arrays wird im Abschnitt 1.6 erläutert.

[2] $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$

Objektmethoden

Objektmethoden enthalten das Schlüsselwort `static` nicht und sind an die jeweilige Instanz gebunden, zu der sie gehören, d.h. dass jede Instanz andere Verhaltensweisen bei dem Aufruf der gleichen Methode aufweisen kann. Objektattribute können auf Klassenattribute, Klassenmethoden, Objektattribute und Objektmethoden zugreifen. Sie können nur mit Hilfe eines Objekts angesprochen werden:

Objekt.Objektmethode

Im Codeabschnitt der Abbildung 1.5 wären folgende Methoden Objektmethoden:

- `public double getRadius()`: Gibt den Radius des Kreises wieder.
- `public double getCircumference()`: Gibt den Umfang des Kreises wieder.
- `public double getArea()`: Gibt den Flächeninhalt des Kreises wieder.

Sie können mit Hilfe eines Objekts aufgerufen werden:

```
1 Circle circle = new Circle(2.5);
2 double radius = circle.getRadius();
3 double circumference = circle.getCircumference();
4 double area = circle.getArea();
```

Abbildung 1.11: Beispielaufruf Objektmethoden - Klasse Circle

Überladen von Methoden

Methoden lassen sich überladen, indem mindestens zwei Methoden vom gleichem Namen innerhalb einer Klasse definiert werden. Der Rückgabetyp und die Parameter können verschieden sein, aber die Parameterlisten müssen verschieden sein. Da eine Methode nicht ausschließlich durch ihren Namen identifiziert wird, sondern auch über die Typen, die Anzahl und die Reihenfolge ihrer Argumente (Parameter) - ihre Signatur, können mehrere Methoden mit demselben Namen in einer Klasse vorhanden sein solange sich die Signaturen unterscheiden.

Um diesen Sachverhalt zu illustrieren können wir zusätzlich zu der Klassenmethode zur Bestimmung der Summen der Umfänge der Kreise in der Abbildung 1.9 eine ähnliche Berechnung definieren:

```
1 public static double getCircumferences(double[] radii) {
2   double circumferences = 0;
3   for (double radius : radii) {
4     circumferences += Circle.PI * 2 * radius
5   }
6   return circumference;
7 }
```

Abbildung 1.12: Überladen von Methoden - Klasse Circle

Wir können nun statt ein `Circle[]` zu erstellen nur die Radien als `double[]` zur Berechnung der Umfänge der Kreise übergeben. Diese Methode unterscheidet sich von der Signatur her von der zuvor definierten Methode, da sich die Parameter der beiden Methoden unterscheiden (`Circle[]` $\neq$ `double[]`).

Beim Aufruf der Methode wird dann, je nach eingegebenen Parametern, die Methode genommen, die zu den Parametern passt. Der Vorteil vom Überladen einer Methode ist, dass man mehrere Methoden mit der gleichen Funktionalität erstellen kann, die für verschiedene Zwecke genutzt werden können. Wie hier zu sehen ist, haben beide `getCircumferences`-Methoden die gleiche Funktionalität. Aber einmal kann man als Eingabe Kreise übergeben und ebenfalls nur die Radien. Je nach Anwendungsfall kann also die passende Methode gewählt werden.

Getter und Setter

Im Zusammenhang zu Methoden sollte man auch als Spezialfall die Setter und Getter-Methoden erwähnen. Es gilt als ein guter Programmierstil, wenn man Attribute auf `private` setzt und Getter und Setter-Methoden zum Zugriff auf die Attribute verwendet. Wie der Name schon sagt, soll mit der Setter-Methode der Wert eines Attributs gesetzt werden können. Mit der Getter-Methode wird dementsprechend der Wert eines Attributs zurückgegeben.

Diese beiden Methoden vereinfachen den Zugriff auf die Attribute und ermöglichen es das Programm besser zu schützen. Wenn beispielsweise der Wert eines Attributs innerhalb von einem bestimmten eingegrenzten Bereich liegen soll, ist es einfacher eine Setter-Methode zu definieren, als überall, wo das Attribut verwendet wird, auf den Bereich zu überprüfen. Außerdem kann man bei Attributen, die nicht verändert werden sollen, keine Setter-Methode definieren und damit den Zugriff darauf regeln.

1.1.4 Konstruktor

Wir haben bereits Konstruktoren in verschiedenen Beispielen, wie in den Abbildungen 1.1, 1.5 und 1.6, gesehen. In diesem Abschnitt wollen wir uns genauer damit befassen, was man genau unter einem Konstruktor versteht.

Ein Konstruktor hat keinen Rückgabewert und sein Name ist gleich dem Namen der Klasse. Konstruktoren werden nur einmal aufgerufen und zwar, wenn man ein neues Objekt erstellt. Man kann neue Objekte einer Klasse mit dem Schlüsselwort `new` instanziieren. Beispielweise, wenn wir ein Objekt der Klasse `Student`, wie in der Abbildung 1.6, erstellen wollen, schreiben wir:

```
Student student = new Student("Max");
```

In die runden Klammern nach dem Namen der Klasse (hier: `Student`), kommen die Werte für die Parameter des Konstruktors. Da der Konstruktor von `Student` den Parameter `name` hat, kommt dort ein Wert, der einen Namen repräsentieren soll, hin.

> **Achtung:** Beim Aufrufen und Zuweisen wird die dann die Adresse des entsprechenden erstellten Objekts zurückgegeben und nicht das Objekt selbst (siehe dazu auch 1.5.3 und 1.5.4).

Konstruktoren werden nicht vererbt und, wenn eine Klasse keinen Konstruktor hat, wird vom Compiler automatisch der leere Konstruktor (no-args constructor - keine Argumente Konstruktor) aufgerufen.

Im Konstruktor können Anweisungen definiert werden, die während der Erstellung eines Objektes gebraucht werden. Beispielsweise können dabei Attributen initialisiert oder bestimmte Methoden angesprochen werden (siehe dazu Abbildung 1.6).

Innerhalb von einem Konstruktor werden oft die Schlüsselwörter `this` und `super` verwendet.

Das Schlüsselwort `this` spricht das aktuelle Objekt an, d.h. wir können mit diesem Schlüsselwort auf die aktuellen Attribute und Methoden des Objektes zugreifen. Wenn ein gleichnamige/r Parameter oder lokale Variable vorhanden ist, kann mithilfe von diesem Schlüsselwort das Attribut von diesem/r Parameter oder lokalen Variable unterschieden werden. Als Beispiel haben wir in der Klasse `Student` gesehen, dass der Konstruktor einen Parameter enthält, wessen Name mit einem Attribut der Klasse übereinstimmt. Um zu unterscheiden, ob das Attribut oder der Parameter angesprochen werden soll, kann das Schlüsselwort `this` verwendet werden.

Wir können zum Codebeispiel `Student` (Abbildung 1.6) das Schlüsselwort `this` an den entsprechenden Stellen hinzufügen und erhalten den folgenden Code:

```
1  public class Student {
2    private static long ID = 0;
3
4    public final long id;
5    private String name;
6
7    public Student(String name) {
8      this.id = ID++;
9      this.name = name;
10   }
11
12   public long getID() {
13     return this.id;
14   }
15
16   public String getName() {
17     return this.name;
18   }
19
20   public void setName(String name) {
21     this.name = name;
22   }
23 }
```

Abbildung 1.13: Verwendung von this - Klasse Student

Das Schlüsselwort `super` spricht die Oberklasse an, d.h. die Klasse von der die jetzige Klasse abgeleitet ist (genaueres dazu wird im Abschnitt 1.2 erläutert). Durch dieses Schlüsselwort können wir den Konstruktor der Oberklasse aufrufen und Attribute und Methoden der Oberklasse ansprechen. Falls der Konstruktor der Oberklasse aufgerufen werden soll, muss es als erste Anweisung im abgeleiteten Konstruktor geschehen. Falls kein Konstruktor der Oberklasse aufgerufen wird, wird der Compiler den Standardkonstruktor der Oberklasse aufrufen, falls einer definiert ist.

1.2 Vererbung

Die Vererbung ist ein grundlegendes Konzept der Objektorientierung, mit welcher Hierarchien in der realen Welt mit Hilfe von Klassen modelliert werden können. Beispielsweise können damit die hierarchischen Beziehungen zwischen Klassen dargestellt werden.

Die Basisklasse stellt eine Verallgemeinerung eines Objekt (*Generalisierung*) mit seinem Verhalten dar, die durch die abgeleitete Klassen spezifiziert werden kann. Innerhalb der abgeleiteten Klassen können also eigene Ausprägungen und ein eigenes Verhalten (*Spezialisierung*) entstehen, wobei die Grundzüge von der Basisklasse vererbt wurden.

Java unterstützt nur Einfachvererbungen und Vererbungen werden mit dem Schlüsselwort `extends` nach dem Klassennamen gekennzeichnet gefolgt von der Klasse, welche die Klasse erweitern soll.

```java
public class Basisklasse {

 public void methodA() {
  ... // Führt Operationen aus
 }

 ...
}
```

```java
public class AbgeleiteteKlasse extends Basisklasse {

 @Override
 public void methodA() {
  ... // Führt neue Operationen aus
 }

 ...

 // Erbt alle Methoden und
 // Attribute der Basisklasse
}
```

Abbildung 1.14: Formaler Aufbau

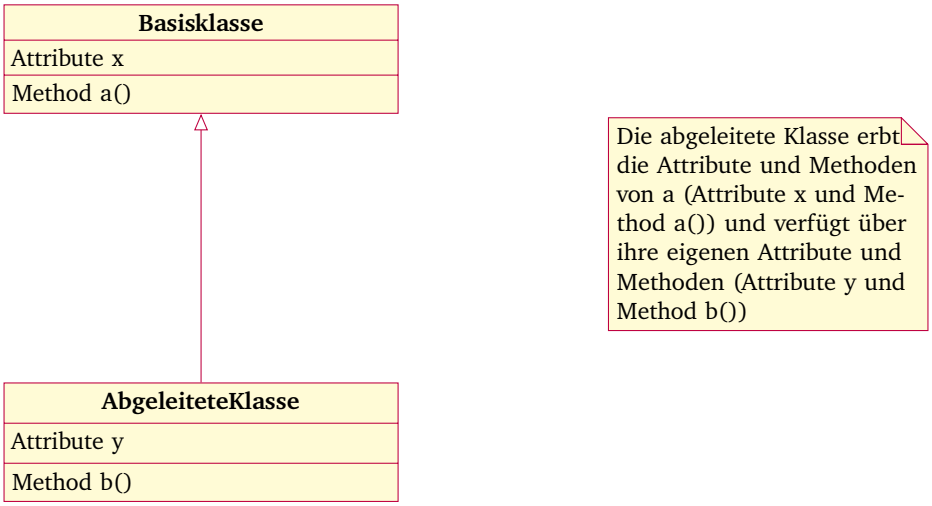

Abbildung 1.15: Darstellung des formalen Aufbaus im UML-Diagramm

Durch eine abstrakte Modellierung kann in der objektorientierten Programmierung, vor allem auf höheren und komplexeren Abstraktionsebenen, eine bessere Strukturierung ermöglicht werden. Dadurch wird es leichter eine weniger fehleranfällige Abbildung der realen Welt zu erschaffen.

Eine wichtige Eigenschaft der Vererbung ist die *Erweiterbarkeit*. Die Modellierung von Abstraktionsebenen ermöglicht es uns bestimmte Strukturen der Klasse wiederzuverwenden und vermeidet die *Redundanz* von gleichen oder ähnlichen Codeabschnitten, die ansonsten mehrfach geschrieben werden müssten. Dadurch wird ebenfalls die *Fehleranfälligkeit* reduziert.

Im folgenden machen wir wieder ein Beispiel: Wir würden gerne Vögel modellieren. Dafür definieren wir als Basisklasse die Klasse `Bird`. Daraus können wir verschiedene Arten von Vögeln ableiten, die gemeinsame Verhalten und Eigenschaften aufweisen.

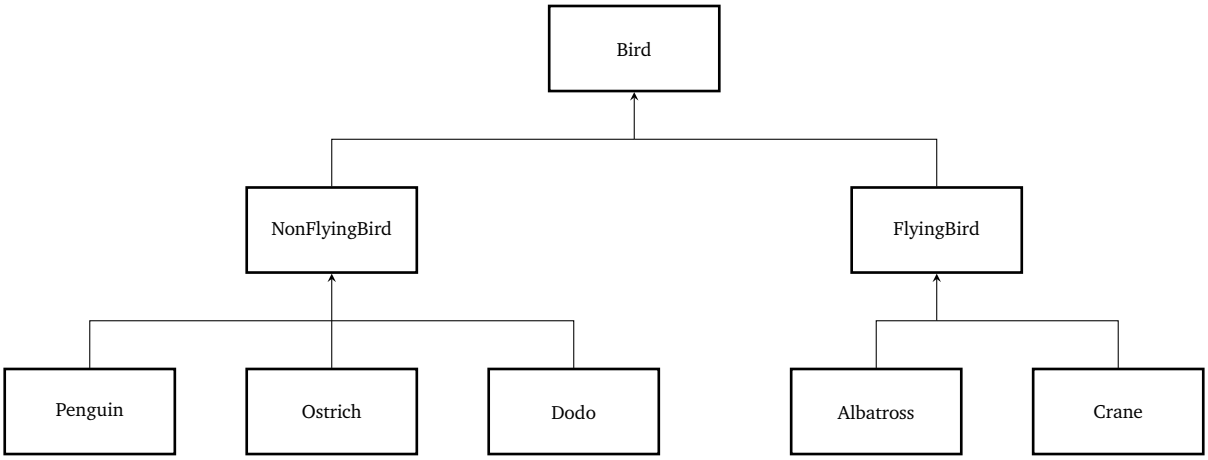

Abbildung 1.16: Typhierarchie - Modellierung von Vögeln

Wir würden also in der Klasse `Bird` alle Verhaltensweisen und Eigenschaften definieren, die alle Vögel gemeinsam haben. In der Klasse `NonFlyingBird` würden wir alle Eigenschaften und Verhaltensweisen definieren, die alle Vögel, die nicht fliegen können, gemeinsam haben usw.

Dadurch ist es uns möglich Wiederholungen und Fehler zu vermeiden. Wir würden nämlich beispielsweise die Methode, die das Fliegen darstellen soll, nicht in `Bird` und `NonFlyingBird` definieren, sondern in `FlyingBird`. Dadurch würden wir sicher stellen, dass einerseits Vögel, wie `Penguin`, nicht auf die Methode zugreifen können. Andererseits müssen wir auch nicht die Methode bei jeder Vogelart neu definieren und Vögel, wie `Crane`, hätten automatisch Zugriff darauf, weil sie von `FlyingBird` abgeleitet sind.

1.2.1 Subtypen

Als Subtypen einer Klasse bezeichnet man

- die Klassen, die entweder direkt `extends` oder indirekt von einer Klasse abgeleitet sind.
- bei einem Interface die Interfaces, die es direkt oder indirekt erweitern, und die Klassen, die es oder eins der erweiterten Interfaces direkt oder indirekt implementieren.
- bei einem Array die Komponententypen, bei denen der Typ ein Subtyp ist (Erklärung zu Arrays siehe 1.6)

Das Gegenstück zu Subtyp ist der Supertyp.

Statischer und dynamischer Typ

Der statische Typ ist der Typ, der bei der Variablendeklaration angegeben wird. Er ist bei der Kompilierung schon bekannt.

Der dynamische Typ ist der Typ des tatsächlichen Objekts, der erst zur Laufzeit bekannt ist. Er kann vom statischen Typ abweichen und ist entweder gleich dem statischen Typ oder ein Subtyp des statischen Typs.

Der formale Aufbau bei einer Variablendeklaration sieht also folgendermaßen aus:

<Statischer Typ> Bezeichner = <Dynamischer Typ>

Um sich das Ganze zu veranschaulichen, betrachten wir den folgenden Codeabschnitt:

```
1  public class Rectangle {
2
3    private final double width;
4    private final double height;
5
6    public Rectangle(double width, double height) {
7      this.width = width;
8      this.height = height;
9    }
10
11   public double getWidth() {
12     return width;
13   }
14
15   public double getHeight() {
16     return height;
17   }
18
19   public double getArea() {
20     return width * height;
21   }
22
23   public double getCircumference() {
24     return 2 * (width + height);
25   }
26
27 }
```

Abbildung 1.17: Vererbung Klasse Rectangle

```
1  public class Square extends Rectangle {
2
3    public Square(double length) {
4      super(length, length);
5    }
6
7    @Override
8    public double getCircumference() {
9      return 4 * getWidth();
10   }
11
12   public double getDiagonalLength() {
13     return Math.sqrt(2 * getWidth() * getWidth());
14   }
15 }
```

Abbildung 1.18: Vererbung Klasse Square

Wir können daraus als Beispiel die folgenden Objekte instanziieren:

```
1 Rectangle rectangle = new Rectangle(22,4);
2 Square square = new Square(5);
3 Rectangle rs = new Square(21);
```

Abbildung 1.19: Vererbung Klasse Rectangle und Square

Wie wir kennen gelernt haben, kann ein Objekt mit dem statischen Typ `Rectangle` entweder `Rectangle` oder ein Subtyp davon als dynamischen Typ haben.

Gehen wir das Beispiel nochmal genauer durch: In Zeile 1 und 2 entspricht der statische Typ dem dynamischen Typ. Aber in Zeile 4 können wir sehen, dass wir als dynamischen Typ einen Subtyp von `Rectangle` genommen haben. Hier wird der Subtyp `Square` der Referenz vom Supertyp `Rectangle` zugewiesen.

Der statische Typ darüber auf welche Attribute und Methoden zugegriffen werden darf und der dynamische Typ welche Implementation einer Methode aufgerufen wird.

> **Information:** Ein zentraler Punkt bei Subtypen ist also, dass man beim dynamischen Typ statt der eigentlichen Klassen ebenfalls ihre Subtypen verwenden kann. Das gilt für die Deklaration von Attributen, Variablen, sowie dem Parameterwert und Rückgabetyp von Methoden.
>
> Beim Parameterwert wird deshalb zwischen dem aktualen Wert (der Parameterwert, der eingesetzt wird) und dem formalen Wert (der in der Definition des Parameters in der Methode steht) unterschieden.

1.2.2 Überschreiben von Methoden

Wir unterscheiden das *Überschreiben* einer Methode vom *Überladen* einer Methode. Das Überladen einer Methode haben wir bereits in Abschnitt 1.1.3 kennen gelernt. Das Überschreiben von Methoden erlaubt es einer abgeleiteten Klasse, eine eigene Implementierung einer von der Basisklasse geerbten Methode zu definieren. Die Unterscheidung, ob die ursprüngliche oder die neue Methode beim Aufruf der überschriebenen Methode verwendet wird, wird anhand des dynamischen Typs (genaueres dazu im Abschnitt 1.2.1) des Objektes getroffen.

Mithilfe vom Überschreiben einer Methode kann man erreichen, dass Methoden, deren Funktionalität in der abgeleiteten Klasse verändert werden soll, einfach überschrieben werden. Gleichzeitig können aber die anderen Methoden der Oberklasse immer noch weiter verwendet werden.

> **Information:** Es gibt also zwei Fälle bei dem Überschreiben einer Methode:
>
> - Ist der dynamische Typ des betrachteten Objekts die abgeleitete Klasse, wird die Methode von der abgeleiteten Klasse verwendet.
> - Ist er hingegen die Basisklasse, so wird die Methode von der Basisklasse verwendet.

Zur Veranschaulichung machen wir uns wieder ein Beispiel. Wir nehmen den Code aus 1.17 und 1.18. In dem Codeabschnitt 1.18 wurde eine Klasse `Square` erstellt, die von der Klasse `Rectangle` abgeleitet ist.

Wie wir sehen können, wurde in `Square` nur die Methode `getCircumference` überschrieben. Würden wir ein Objekt vom statischen Typ `Rectangle` und dynamischen Typ `Square` haben und `getCircumference` aufrufen, würde die Methode, die in `Square` definiert wurde, aufgerufen werden. Bei einem statischen und dynamischen Typ von `Rectangle` würde die Methode `getCircumference` von `Rectangle` genommen werden.

Überschriebene Methoden können mit der Annotation `@Override` versehen werden. Die Annotation `@Override` gibt an, dass die Methode der untergeordneten Klasse die Methode der Basisklasse überschreibt. Aus zwei Gründen ist die Annotation `@Override` nützlich:

- Wenn die mit der Annotation versehene Methode nichts überschreibt, gibt der Compiler eine Warnung aus. Dadurch können Fehler vermieden werden
- Es kann helfen, den Quellcode lesbarer zu machen.

1.2.3 Methodentabelle

Wir haben bereits kennen gelernt, dass der dynamische Typ eines Objektes eine Rolle dabei spielt, welche Implementation einer Methode aufgerufen wird. Aber was genau versteht man unter „Welche Implementation einer Methode aufgerufen wird“?

Jedes Objekt einer Klasse enthält einen Verweis auf ein anonymes Objekt mit verborgenen Informationen. Dieses anonyme Objekt wird einmal in einem Programm pro Klasse eingerichtet. Die Informationen beziehen sich auf die Klasse selbst, ihre Methoden und ihre Attribute. Außerdem enthält das anonyme Objekt auch die Methodentabelle.

Im Folgenden wollen wir die Abbildung 1.19 und die entsprechenden Definitionen der dort verwendeten Klassen zur Veranschaulichung des Speichers und der Methodentabelle nutzen.

Zuerst schauen wir an, wie die verborgenen Informationen im Speicher aussehen:

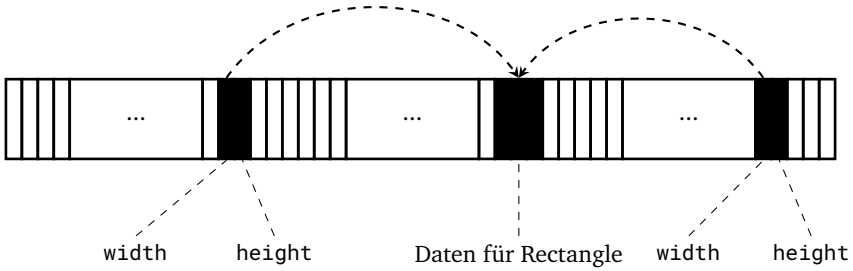

Abbildung 1.20: Anonymes Objekt zur Klasse `Rectangle`

Wir sehen hier zwei Objekte von `Rectangle`, die jeweils die Attribute `width` und `height` haben. Jedes von diesen zwei Objekten hat dabei einen Verweis auf die verborgenen Informationen, die hier als „Daten für `Rectangle`“ bezeichnet werden. Im Allgemeinen soll die Abbildung also zeigen, dass, wenn man ein Objekt von `Rectangle` erstellt, es automatisch einen Verweis auf die verborgenen Informationen bekommt. Jedes Objekt vom Typ `Rectangle` kann somit auf diese Informationen zugreifen. Das Beispiel lässt sich auf alle möglichen Klassen, nicht nur auf `Rectangle`, übertragen.

Mithilfe der Methodentabelle, die zu den verborgenen Informationen gehört, kann man sehen, welche Implementation einer Methode für ein gegebenes Objekt verwendet wird.

Wir wollen die Funktionsweise von der Methodentabelle anhand des gegebenen Beispiels kennen lernen:

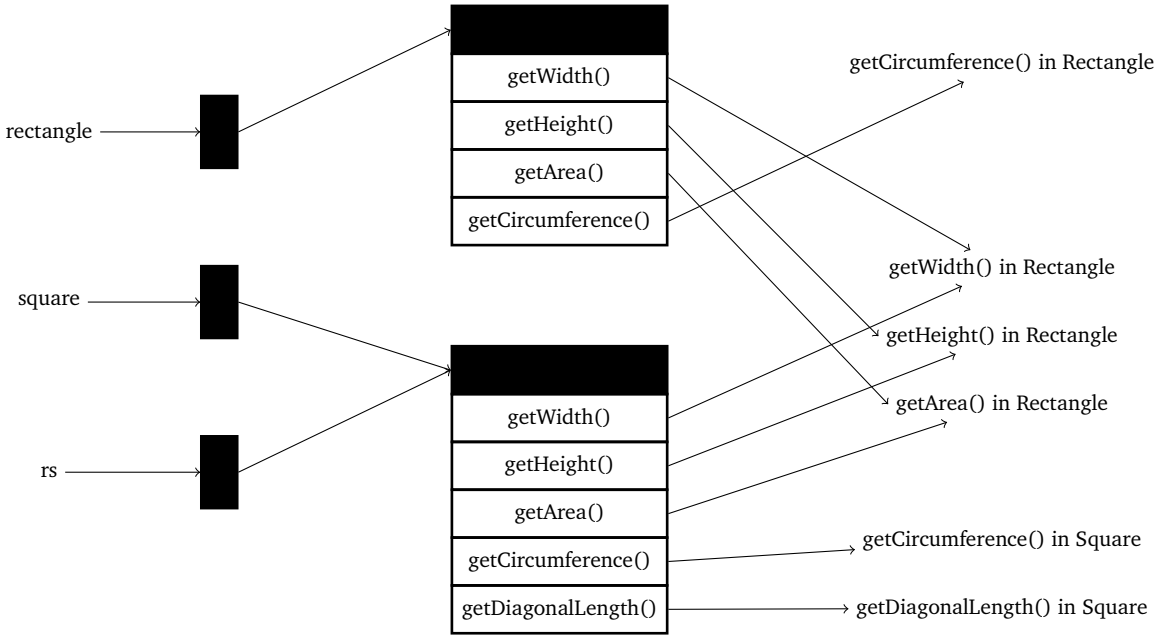

Abbildung 1.21: Methodentabelle bezüglich des Codeabschnitt aus der Abbildung 1.19

Wir wollen uns die einzelnen Variablen anschauen und herausfinden, welche Methoden welcher Klassen bei welcher Variablen aufgerufen werden.

- Objekt `Rectangle` hat den dynamischen und statischen Typ `Rectangle`, deshalb ist es klar, dass die Implementation `getWidth()`, `getHeight()`, `getArea()` und `getCircumference()` von `Rectangle` verwendet werden.
- Objekt `Square` hat den statischen und dynamischen Typ `Square`, deshalb wird `getCircumference` und `getDiagonalLength()` von `Square` genommen. Die Methoden `getWidth()`, `getHeight()` und`getArea()` werden von `Rectangle` verwendet, da es keine Implementation von diesen Methoden in `Square` gibt.
- Objekt `rs` hat den statischen Typ `Rectangle` und den dynamischen Typ `Square`. Wegen dem dynamischen Typ `Square` wird die Implementation `getCircumference` von `Square` genommen, obwohl das Objekt den statischen Typ `Rectangle` hat. Natürlich wird wieder `getWidth()`, `getHeight()` und`getArea()` von `Rectangle` verwendet. Achtung: Auf die Methode `getDiagonalLength()` kann aufgrund von dem statischen Typs nicht zugegriffen werden!

Casting

Wenn eine Variable von einem bestimmten Typ ist, aber man einen anderen Typ verwenden will, wie z.B. einen abgeleiteten Typ, ist es möglich die Variable auf den entsprechenden Typ zu „casten“.

Casting bedeutet so etwas wie Typumwandlung und kann entweder explizit oder implizit durchgeführt werden. Beim impliziten Casting wird der Typ nicht angegeben, sondern die Typumwandlung geschieht automatisch. Implizites Casting ist immer dann möglich, wenn das Casting unbedenklich ist, d.h. der Typ in den anderen Typ ohne Datenverlust umgewandelt werden kann (bspw. `short` zu `int`). Beim expliziten Casting muss der Typ, in den umgewandelt werden soll, explizit angegeben werden.

Der Vorteil und der Sinn vom Casting lässt sich an dem folgenden Beispiel zeigen: Nehmen wir an wir haben den statischen Typ einer Klasse `Rectangle` genommen, wollen aber auf die Methoden, die erst in der Klasse `Square` definiert werden, zugreifen. Das geht nur, wenn wir vorher die zugehörige Variable downcasten, d.h. in den Typ `Square` umwandeln. Das können wir durch explizites Casting erreichen.

Um eine Variable explizit zu einem Typ zu casten, können wir das folgende Schema verwenden

(Typ des Casting) Objekt

In unserem konkreten Beispiel würde das so aussehen, wenn wir auf die Methode `getDiagonalLength()` von `Square` zugreifen wollen:

((Square) rs).getDiagonalLength();

1.3 Interfaces und abstrakte Klassen

1.3.1 Interface

Ein Interface ist eine Schnittstelle, die trennt, *was* eine Klasse tut, und *wie* die Klasse es tut, und in einem Interface können keine Objekte instanziiert werden.

Das Interface stellt einen Vertrag dar: Es setzt fest, was ein Kunde erwarten kann, indem im Vorhinein die Funktionalitäten beschrieben werden, die erfüllt werden sollen, und genaue Methoden benannt werden. Der Kunde weiß damit, welche Methoden und Funktionalitäten eine Klasse, die das Interface implementiert, haben muss. Außerdem lässt das Interface dem Entwickelnden die Freiheit, wie diese Erwartung durch die eigentliche Implementierung erfüllt wird. Das heißt, dass die genaue Implementierung der beschriebenen Methoden durch den Entwickelnden in der das Interface implementierenden Klasse erfolgt und nicht direkt im Interface selbst ersichtlich ist. Interfaces erlauben damit die Verwendung von Funktionalitäten, ohne, dass die eigentliche darunterliegende Implementierung bekannt ist.

Zur Visualisierung kann man sich Interfaces als eine Art Schablone vorstellen. Die Schablone gibt eine Vorlage, wie das Produkt am Ende aussehen soll. Allerdings gibt sie nicht alles vor, sondern ihr Inneres ist frei gestaltbar, solange die Vorgaben der Schablone eingehalten werden.

Im Interface ist alles implizit `public`, weshalb der Zugriffsmodifikator in Interface weggelassen werden kann. Zusätzlich sind alle Methoden implizit `abstract` und das Schlüsselwort `abstract` sagt aus, dass die Methode nicht implementiert ist. Es kann hier ebenfalls weggelassen werden. Es gibt allerdings einige Ausnahmen:

- Seit Java 8 können Objektmethoden, welche mit dem Schlüsselwort `default` versehen sind, implementiert werden. Dadurch können die Entwickelnden den Interfaces neue Methoden hinzufügen ohne die Klassen zu beeinflussen, die diese Interfaces implementieren.
- Seit Java 9 können Methoden können mit dem Zugriffsmodifikator `private` implementiert werden. Diese privaten Methoden verbessern die Wiederverwendbarkeit von Code innerhalb von Schnittstellen. Das passiert dadurch, dass man Code in private Methoden auslagern kann, die dann von außen nicht sichtbar sind und bspw. in default-Methoden verwendet werden können.

Außerdem können in einem Interface nur Objektmethoden deklariert werden.

Da ein Interface zustandslos ist, können keine Objektattribute definiert werden, sondern nur Klassenkonstanten. Also können bei der Definition von Klassenkonstanten die Schlüsselwörter `static final` weggelassen werden, da sie sowieso implizit verwendet werden.

Eine Klasse kann ein Interface oder mehrere Interfaces mit dem Schlüsselwort `implements` (im Klassenkopf) implementieren. Falls mehrere Interfaces implementiert werden sollen, dann werden die Namen mit einem Komma voneinander getrennt hintereinander notiert. Eine Klasse muss alle nicht implementierten Methoden der implementierten Interfaces implementieren, sonst muss die Klasse als abstrakt gekennzeichnet werden.

Ein Interface kann auch ein oder mehrere Interfaces erweitern. Das wird mit dem Schlüsselwort `extends` gekennzeichnet und die entsprechenden Interfaces werden mit einem Komma getrennt nacheinander aufgeschrieben.

Als Beispiel definieren wir uns zwei Interface:

- Das Interface `Swimming` bietet die Möglichkeit eine bestimmte Distanz zu schwimmen.
- Das Interface `Flying` bietet die Möglichkeit eine bestimmte Distanz zu fliegen. Außerdem beinhaltet es eine Klassenkonstante `SPEED`. Mithilfe von dieser Konstante ist es möglich festzulegen, dass einige Objekte mehr Distanz zurücklegen können, weil sie eine höhere Geschwindigkeit haben.

Jetzt können wir einen Schwan modellieren, der schwimmen und fliegen kann. Der Zugriffsmodifikator bei der Methode `fly` kann weggelassen werden, da standardmäßig alles im Interface `public` ist. Wir haben ihn lediglich zu Demonstrationszwecken beibehalten.

```
1  public interface Swimming {
2
3    void swim(double distance);
4  }
5
6  public interface Flying {
7
8    double SPEED = 2;
9
10   public abstract void fly(double distance);
11 }
12
13 public class Swan implements Swimming, Flying {
14
15   private double distance = 0;
16
17   @Override
18   public void swim(final double distance) {
19     this.distance += distance;
20   }
21
22   @Override
23   public void fly(final double distance) {
24     this.distance += Flying.SPEED * distance;
25   }
26 }
```

Abbildung 1.22: Beispiel Interface - Flying und Swimming

1.3.2 Abstrakte Klassen

Abstrakte Klassen sind Klassen, welche mit dem Schlüsselwort `abstract` gekennzeichnet werden. Wie beim Interface können ebenfalls keine Objekte von abstrakten Klassen instanziiert werden. Aber im Gegensatz zum Interface dürfen Methoden in abstrakten Klassen auch implementiert sein ohne, dass es sich automatisch um `default`-Methoden handelt. Nicht implementierte Methoden in abstrakten Klassen werden mit dem Schlüsselwort `abstract` gekennzeichnet. Es können alle Arten von Attributen definiert werden und zusätzlich sind auch alle Zugriffsmodifikatoren erlaubt.

Abstrakte Klassen erlauben es Vorlagen mit Zuständen und Funktionalitäten für Unterklassen zur Verfügung zu stellen, welche dann genutzt oder spezifiziert werden können. Sie werden oft verwendet, um Eigenschaften und Funktionalitäten einer allgemeinen Typgruppe zu definieren, die von den abgeleitete Klassen weiter spezifiziert werden. Ein zusätzlicher Vorteil von abstrakten Klassen ist, dass keine Objekte von abstrakten Klassen erstellt werden können. Dadurch können „Schemas“ erstellt werden statt vollständig funktionsfähige implementierte Klassen.

> **Information:** Ein Interface kann beliebig viele Interfaces erweitern und abstrakte Klassen können nur eine Klasse erweitern (Einfachvererbung). Des Weiteren gilt eine Klasse als abstrakt, sobald sie eine nicht implementierte Methode vorhanden ist.

Zur Verdeutlichung der Idee von abstrakten Klassen verwenden wir den Codeabschnitt aus der Abbildung 1.22 und modifizieren ihn. Wir definieren uns dazu eine abstrakte Klasse `FlyingBird`, welche fliegende Vögel darstellen soll. Sie enthält zwei Attribute: Die bisher zurückgelegte Strecke, welche in Unterklassen sichtbar sein soll, und den Namen des entsprechenden Vogels. Diese abstrakte Klasse implementiert außerdem das Interface `Flying`. Aber sie muss nicht unbedingt alle Methoden des Interface implementieren, da die Klasse abstrakt ist. Wir können auch zusätzliche Methoden implementieren, wie bspw. `getName()`. Mithilfe der abstrakten Klasse als Vorlage können wir nun konkrete fliegende Vögel, wie `Swan`, erstellen.

```java
public abstract class FlyingBird implements Flying {

  protected double distance = 0;
  private final String name;

  public FlyingBird(String name) {
    this.name = name;
  }

  @Override
  public abstract void fly(final double distance);

  public String getName() {
    return name;
  }
}

public class Swan extends FlyingBird implements Swimming {

  public Swan(String name) {
    super(name);
  }

  @Override
  public void swim(final double distance) {
    this.distance += distance;
  }

  @Override
  public void fly(final double distance) {
    this.distance += Flying.SPEED * distance;
  }
}
```

Abbildung 1.23: Beispiel abstrakte Klasse - FlyingBird

1.4 Scope

Der Begriff Scope wird meistens in Bezug auf Methoden und Variablen verwendet und bezeichnet den Gültigkeitsbereich, in dem auf die Methoden und Variablen zugegriffen werden kann. Wenn eine Methode oder Variable außerhalb ihres Gültigkeitsbereiches aufgerufen wird, kommt es zu einer Fehlermeldung. Außerdem dürfen (mit einigen Ausnahmen) innerhalb von dem gleichen Scope keine gleichen Variablen oder Methoden definiert werden.

> **Information:** Die besagten Ausnahmen sind einmal Attribute und lokale Variablen und einmal überladene Methoden.
>
> Eine lokale Variable darf den gleichen Namen wie ein Attribut haben, da man beide durch das Schlüsselwort `this` voneinander unterscheiden kann.
>
> Überladene Methoden kann man durch die Signaturen voneinander unterscheiden, obwohl sie den gleichen Namen haben.

1.4.1 Attribute vs Lokale Variablen und Konstanten

Als lokale Variablen und Konstanten werden Variablen bzw. Konstanten, die innerhalb von einer Methode definiert werden, Parameter von Methoden und Variablen, die innerhalb von `for`-Schleifen definiert werden, bezeichnet.

Im Gegensatz dazu sind Attribute *globale* Variablen, denn man kann auf Attribute überall in der Klasse zugreifen. Außerhalb von der Klasse wird der Zugriff auf Attribute durch Zugriffsmodifikatoren geregelt. Zugriffsmodifikatoren haben Sie bereits in Abschnitt 1.1.1 kennen gelernt. Sie können dort herauslesen, wie der Zugriff je nach Zugriffsmodifikator für Attribute geregelt ist.

Zur Veranschaulichung der Unterschiede zwischen lokalen Variablen und Attributen machen wir ein kleines Beispiel. Wir definieren eine Klasse, deren Aufgabe es ist beim Aufruf der Methode `countUp` den Wert einer lokalen Variable und einer globalen Variable um einen gegebenen Wert hochzuzählen. Außerdem kann durch die Methode `getCount()` die gesamte Anzahl an Werten zurückgegeben werden, mit denen `countUp` aufgerufen wurde.

```
1  public class Counter{
2
3    private int count;
4
5    public Counter(){
6      count = 0;
7    }
8
9    public int countUp(int steps){
10     int localCount = 0;
11     for(int i = 0; i < steps; i++){
12       localCount++;
13       count++;
14     }
15     return localCount;
16   }
17
18   public int getCount(){
19     return count;
20   }
21 }
```

Abbildung 1.24: Beispiel zur Veranschaulichung des Unterschieds zwischen lokalen Variablen und Attributen

Zuerst sehen wir uns das Attribut `count` an. Da wir den Zugriffsmodifikator `private` verwendet haben, können wir zwar überall in der Klasse `count` darauf zugreifen, aber nicht außerhalb von dieser Klasse. Also können wir `count` ohne weiteres in dem Konstruktor der Klasse, sowie in den Methoden `countUp` und `getCount()` verwenden.

Als Nächstes betrachten wir die lokalen Variable `localCount` und `steps` . Weil wir sie innerhalb der Methode bzw. als Parameter der Methode definiert haben, können wir auch nur innerhalb von der Methode `localCount` darauf zugreifen. Wenn wir versuchen würden außerhalb von dieser Methode darauf zuzugreifen, würden wir eine Fehlermeldung bekommen.

Als Letztes betrachten wir `i`. Dabei handelt es sich wieder um eine lokale Variable. Jedoch wurde `i` im Schleifenkopf der `for`-Schleife definiert. Deshalb kann man auf diese Variable auch nur innerhalb von der Schleife zugreifen. Würden wir versuchen außerhalb von der Schleife darauf zuzugreifen, würden wir eine Fehlermeldung bekommen.

> **Information:** Wie wir bereits kennen gelernt haben hilft das Schlüsselwort `this`, um lokale Variablen von Attributen mit gleichem Namen zu unterscheiden. Wenn man `this` vor einen Variablennamen schreibt, wird automatisch immer das Attribut genommen. Ohne das `this` wird entsprechend immer die lokale Variable verwendet.

1.4.2 Methoden

Der Scope von Methoden wird ebenfalls durch die Zugriffsmodifikatoren geregelt. Innerhalb von einer Klasse kann man auf alle Methoden von der Klasse zugreifen. Wenn man auf eine Methode außerhalb von der Klasse zugreifen will, hängt es von dem Zugriffsmodifikator ab, ob es möglich ist. Genaueres zu Zugriffsmodifikatoren steht in 1.1.1.

1.5 Primitive Datentypen und Referenztypen

1.5.1 Wertgleichheit und Objektidentität

Im Zusammenhang zu primitiven Datentypen und Referenztypen ist es wichtig, dass wir die Begriffe der Objektidentität und der Wertgleichheit einführen.

Objektidentität heißt, dass mehrere Referenzen auf dasselbe Objekt verweisen. Wertgleichheit bedeutet, dass mehrere Referenzen auf zwei Objekte mit demselben Inhalt verweisen. Was das genau im Zusammenhang mit Referenztypen bedeutet, werden wir in den folgenden Abschnitten nochmal aufgreifen.

Wir können uns aber nochmal merken, dass Objektidentität bedeutet, dass man zwei Mal das gleiche Objekt hat. Es gibt also keine Kopie des Objekts, da beide Referenzen auf das gleiche Objekt verweisen. Bei Wertgleichheit müssen die Objekte nicht unbedingt gleich sein. Also ist eine Kopie eines Objekts wertgleich mit dem ursprünglichen Objekt, aber es muss nicht dasselbe Objekt sein.

1.5.2 Primitive Datentypen

Ein primitiver Typ ist von der Sprache vordefiniert und wird durch ein reserviertes Schlüsselwort gekennzeichnet. Sie können jeweils nur einen Wert speichern und zuweisen, d.h. wenn wir einer Variablen eines primitiven Datentyps einen neuen Wert zuweisen, wird der bisher gespeicherte Wert überschrieben.

Wir können wegen der obigen Definition nicht die Begriffe der Objektidentität und Wertgleichheit auf primitive Datentypen beziehen, da es keine Referenzen gibt. Aber wir können sagen, dass Zuweisen und Kopieren bei primitiven Datentypen als synonym zu betrachten sind. Denn durch eine Zuweisung wird der tatsächliche Wert von primitiven Datentypen in den neuen Speicherplatz kopiert und nicht die Adresse von dem Wert (, dazu später auch ein Beispiel bei Abbildung 1.27).

Die primitiven Datentypen in Java sind

- `boolean`, `byte`, `short`, `int`, `long`, `float`, `double` und `char`.

> **Information:** Mehr Informationen zu primitiven Datentypen finden Sie unter folgendem Verweis:
>
> `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html`

1.5.3 Referenztypen

Als Referenztypen bezeichnet man Objekte von Klassen, Arrays und Enums. Bei Referenztypen muss zuerst ein Speicherplatz für das jeweilige Objekt allokiert [3] werden, deshalb muss man bei der Erstellung das Schlüsselwort `new` verwenden. Wie der Name „Referenztyp“ schon sagt, wird im Speicher eine Adresse bzw. eine Referenz auf den Wert des Objekts abgelegt und nicht der eigentliche Wert, wie bei primitiven Datentypen. Da man nun einmal die gespeicherte Adresse hat und einmal den Wert, auf den die Adresse zeigt, bedeutet bei Referenztypen Wertgleichheit und Objektgleichheit nicht unbedingt dasselbe.

[3] Allokation = Reservierung von Hauptspeicher

1.5.4 Vergleich zwischen Referenztypen und primitiven Datentypen

Im folgenden wollen wir einige Beispiele betrachten, die den Unterschied zwischen primitiven Datentypen und Referenztypen verdeutlichen sollen.

Nehmen wir an, dass wir eine Klasse `Rectangle` definieren wollen, die ein Rechteck darstellen soll. Diese Klasse soll zwei Attribute haben - die Höhe und die Breite des Rechtecks.

```
1 public class Rectangle {
2
3   double width;
4   double height;
5 }
```

Abbildung 1.25: Vereinfachtes Modell - Klasse Rectangle

Wie wir sehen können, sind die Typen der Attribute `width` und `height` primitive Datentypen und der Typ eines Objekts von `Rectangle` wäre ein Referenztyp.

Angenommen wir erstellen jetzt ein Objekt von `Rectangle` namens `r`. Dann sieht das Ganze so im Speicher aus:

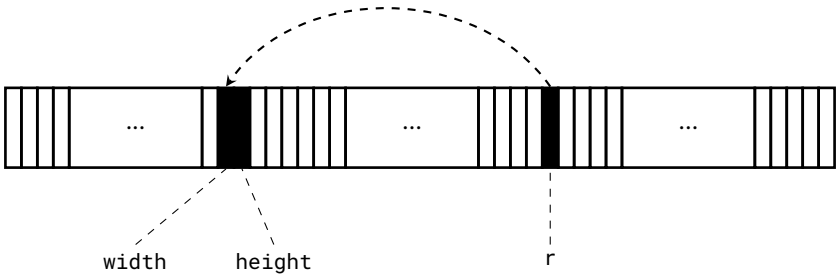

Abbildung 1.26: Abstrakte Visualisierung des Speicherplatzes eines `Rectangle`-Objekts - `Rectangle r = new Rectangle();`

Wie wir sehen können, wird der Wert bei `width` und `height` direkt gespeichert, während `r` auf den Speicherort des Objekts verweist, also die Adresse speichert.

1.5.5 Wertgleichheit vs. Objektgleichheit

Primitive Datentypen

Wie bereits erwähnt wurde, ist Zuweisen und Kopieren bei primitiven Datentypen äquivalent.

Um diese Tatsache zu veranschaulichen, sehen wir uns das folgende Codebeispiel an:

```
1 double width = 9;
2 double newWidth = 8;
3 newWidth = width;
```

Abbildung 1.27: Einfaches Beispiel zu Zuweisungen bei primitiven Datentypen

Das sieht dann folgendermaßen im Speicher aus:

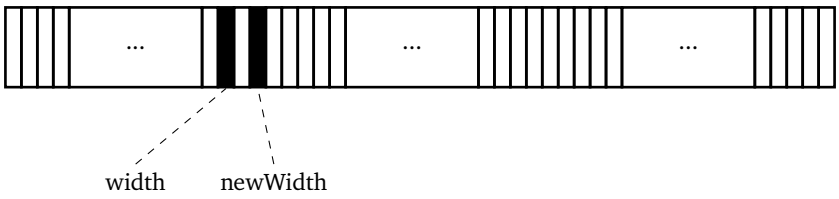

Abbildung 1.28: Abstrakte Visualisierung des Speicherplatzes von Beispiel 1.27

Wenn wir den Codeauschnitt aus der Abbildung 1.27 ausführen, würden wir zuerst die Variablen `width` und `newWidth` erstellen und im Speicher ablegen. Die jeweiligen Werte in den Speicherplätzen können bei primitiven Datentypen über den Namen der jeweiligen Variablen herausgelesen werden.

Zeile im Code	`width`	`newWidth`
1	9	nicht definiert
2	9	8
3	9	9

Tabelle 1.3: Trace Table für Abbildung 1.27

Bei der Zuweisung in Zeile 3 würden wir also auf den Wert in `width` zugreifen, ihn herauslesen und dann in `newWidth` abspeichern. Der Zuweisungsoperator = würde also effektiv den gespeicherten Wert übertragen und keinen Einfluss auf den Speicherplatz haben. Deshalb bedeutet Zuweisen und Kopieren bei primitiven Datentypen dasselbe.

Ein weiterer Punkt ist, dass primitive Datentypen mithilfe von `==` auf Wertgleichheit getestet werden können. Wenn wir also beispielsweise bei dem Codeauschnitt aus der Abbildung 1.27 in der nächsten Zeile auf `newWidth == width` testen würden, würde das Ergebnis `true` sein, da beide Variablen den Wert 9 speichern.

Referenztypen

Im Gegensatz dazu ist bei Referenztypen Objektgleichheit und Wertgleichheit nicht unbedingt dasselbe.

Um diese Tatsache zu veranschaulichen, sehen wir uns das folgende Beispiel an:

```
1 Rectangle r = new Rectangle();
2 Rectangle q = new Rectangle();
3 Rectangle s = r;
4 q.width = r.width;
5 q.height = r.height;
```

Abbildung 1.29: Einfaches Beispiel zu Zuweisungen bei Referenztypen

Wir haben drei Objekte vom Typ `Rectangle` erstellt. Dabei haben wir einmal einem Objekt ein anderes Objekt mit dem Zuweisungsoperator zugewiesen und einmal haben wir ein neues Objekt erstellt und nur die Werte der Attribute mit dem Zuweisungsoperator zugewiesen.

Das sieht dann folgendermaßen im Speicher aus:

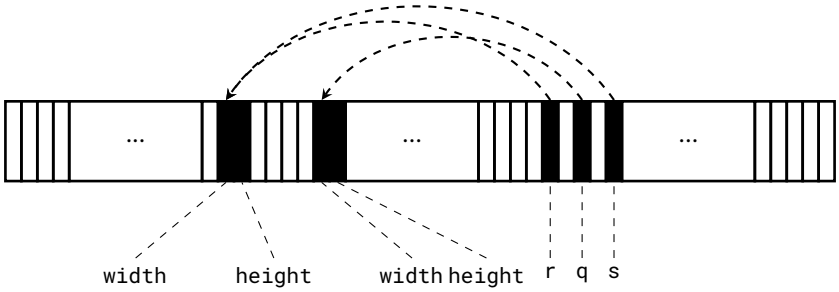

Abbildung 1.30: Abstrakte Visualisierung des Speicherplatzes von Beispiel 1.29

Wie wir hier klar sehen können, referenzieren `r` und `s` auf dasselbe Objekt. Wenn wir also die Attribute von `r` ändern, werden auch automatisch die Attribute von `s` geändert. Hier handelt es sich um Objektgleichheit. Im Gegensatz dazu verweisen `r` und `q` auf verschiedene Objekte. In der Abbildung 1.29 haben wir gesehen, dass wir die Werte der Attribute zugewiesen haben. Die Werte der Attribute in `r` und `q` sind also gleich, während die Objekte verschieden sind. Das bezeichnet man als Wertgleichheit. Natürlich gilt Wertgleichheit auch bei `r` und `s`. Dennoch gilt bei `r` und `q` keine Objektgleichheit, wie es bei `r` und `s` galt.

Wir hatten erwähnt, dass man bei primitiven Datentypen mit `==` auf Wertegleichheit testen kann. Bei Referenztypen kann man `==` nur auf Objektgleichheit testen, da eine Referenz und kein Wert gespeichert wird. Wenn wir explizit auf Wertegleichheit testen wollen, müssen wir den `equals`-Operator verwenden.

Information: Wir haben den Unterschied zwischen Objektgleichheit und Wertgleichheit bei Referenztypen kennen gelernt. Das ist der Grund dafür, dass man immer, wenn man ein Objekt von einem Referenztyp kopieren will, nicht das gesamte Objekt zuweisen sollte. Stattdessen sollte man ein neues Objekt erstellen und den Inhalt des bisherigen Objekts einzeln rüberkopieren. Dabei muss man natürlich aufpassen, dass Attribute auch Referenztypen sein können.

1.6 Arrays

Ein Array repräsentiert einen Block einer festen Größe von reserviertem Speicher eines gleichen Komponententyps (Datentyp der Elemente). Arrays gehören zu den Referenztypen, aber können auch durchaus primitive Datentypen als Komponenten speichern. Die Größe des Arrays muss in Java bei der Erstellung eines Arrays angegeben werden und der Komponententyp eines Arrays muss entweder gleich oder ein Subtyp des angegebenen Typen sein.

Nach folgendem Schema kann ein Array definiert werden:

```
Datentyp[] Bezeichner = new Datentyp[Größe[4]];
```

Es gibt auch eine Kurzform, falls die Elemente im Array im vorhinein bekannt sind:

```
Datentyp[] Bezeichner = {Element1, Element2, ...};
```

Zur Veranschaulichung kann man sich ein Array wie ein Bücherregal vorstellen. Die Größe des Arrays repräsentiert die Anzahl der Fächer im Bücherregal und in jedes Fach kann (in einem eindimensionalen Array) genau ein Buch gelegt werden. Es ist auch nicht möglich mehr Bücher ins Regal zu stellen als es Fächer gibt.

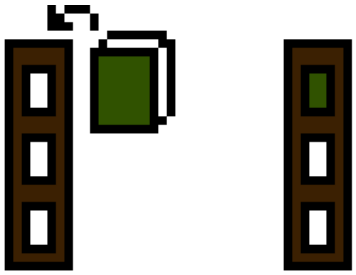

Abbildung 1.31: Array als Bücherregal

[4] Die Größe ist eine natürliche Zahl.

Wir möchten nun, wie in 1.31 ein Bücherregal mit drei Fächern erstellen. Dazu benötigen wir zuerst eine Klasse, die Bücher darstellen soll:

```
1  public class Book {
2
3    private final String title;
4    private final String author;
5    private double cost;
6
7    public Book(String title, String author, double cost) {
8      this.title = title;
9      this.author = author;
10     this.cost = cost;
11   }
12
13   public String getTitle() {
14     return title;
15   }
16
17   public String getAuthor() {
18     return author;
19   }
20
21   public double getCost() {
22     return cost;
23   }
24
25   public void setCost(final double cost) {
26     this.cost = cost;
27   }
28 }
```

Abbildung 1.32: Klasse Book

Nun können wir ein Bücherregal erstellen:

```
1 Book[] bookshelf = new Book[3];
2 bookshelf[0] = new Book("Der Da Vinci Code", "Dan Brown", 12.0);
3 bookshelf[1] = new Book("Harry Potter und die Heiligtümer des Todes", "Joanne K. Rowling", 13.99);
4 bookshelf[2] = new Book("Der Hobbit", "J. R. R. Tolkien", 18.0);
```

Abbildung 1.33: Bücherregal als Array

Da wir aus der Abbildung 1.33 wissen, welche Bücher das Array enthalten soll, könnten wir als Alternative auch die Kurzform verwenden:

```
Book[] bookshelf = {
    new Book("Der Da Vinci Code", "Dan Brown", 12.0),
    new Book("Harry Potter und die Heiligtümer des Todes", "Joanne K. Rowling", 13.99),
    new Book("Der Hobbit", "J. R. R. Tolkien", 18.0)
};
```

Abbildung 1.34: Bücherregal als Array - Kurzform

Natürlich kann der Komponententyp eines Array ebenfalls wieder ein Array sein und man kann Arrays auch weiter beliebig viel verschachteln. Damit wäre ein zweidimensionales Array (Datentyp[][]) ein Array, wessen Komponententyp eindimensionale Array sind. Bei einem dreidimensionalen Array (Datentyp[][][]) sind die Komponententyp zweidimensionale Arrays und deren Komponententyp sind jeweils eindimensionale Arrays.

> **Information:** Beachten Sie, dass die Größe der Arrays innerhalb der Komponenten nicht unbedingt alle gleich sein müssen!

Nehmen wir uns als Beispiel wieder das Bücherregal. Wenn wir ein zweidimensionales Array betrachten, hat das Bücherregal wieder eine feste Anzahl an Fächern. Diese Anzahl wird vom äußeren Array bestimmt. Nun können wir aber die Größe der einzelnen Fächer variieren. Dadurch kann beispielsweise das erste Fach maximal ein Buch enthalten, das zweite Fach aber zwei Bücher etc.

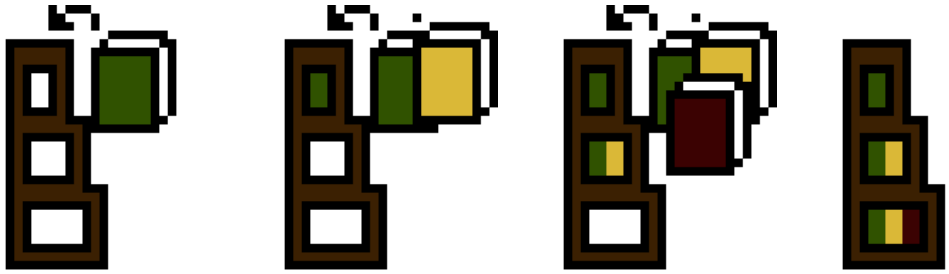

Abbildung 1.35: Verschachteltes Array als Bücherregal

Als Code könnte wir uns das folgendermaßen vorstellen:

```
Book[][] bookshelf = new Book[3][];
bookshelf[0] = new Book[1];
bookshelf[0][0] = new Book("Der Da Vinci Code", "Dan Brown", 12.0);

bookshelf[1] = new Book[2];
bookshelf[1][0] = new Book("Harry Potter und der Stein der Weisen", "Joanne K. Rowling", 8.99);
bookshelf[1][1] = new Book("Harry Potter und die Heiligtümer des Todes", "Joanne K. Rowling", 13.99);

bookshelf[2] = new Book[3];
bookshelf[2][0] = new Book("Der Schmied von Großholzingen", "J. R. R. Tolkien", 14.95);
bookshelf[2][1] = new Book("Der Herr der Ringe", "J. R. R. Tolkien", 12.0);
bookshelf[2][2] = new Book("Der Hobbit", "J. R. R. Tolkien", 18.0);
```

Abbildung 1.36: Bücherregal als verschachteltes Array

Da wir auch hier im vorhinein wissen, welche Bücher im Array gespeichert werden sollen, können wir wieder die Kurzform verwenden:

```
Book[][] bookshelf = {
    {
        new Book("Der Da Vinci Code", "Dan Brown", 12.0)
    },
    {
        new Book("Harry Potter und der Stein der Weisen", "Joanne K. Rowling", 8.99),
        new Book("Harry Potter und die Heiligtümer des Todes", "Joanne K. Rowling", 13.99)
    },
    {
        new Book("Der Schmied von Großholzingen", "J. R. R. Tolkien", 14.95),
        new Book("Der Herr der Ringe", "J. R. R. Tolkien", 12.0),
        new Book("Der Hobbit", "J. R. R. Tolkien", 18.0)}
};
```

Abbildung 1.37: Bücherregal als verschachteltes Array - Kurzform

Information: Intern im Speicher bezieht sich die Adresse bei einem Array auf die Speicheradresse des ersten Elements (Basisadresse). Die nachfolgenden Elemente können von der Basisadresse addiert mit einem Offset angesprochen werden. Dieser Offset entspricht der Speichergröße des Komponententyps.

Beispielsweise um das dritte Element eines Arrays anzusprechen, verwenden wir:

$$\text{Basisadresse} + 2 \cdot \text{Offset}$$

(Die Basisadresse entspricht dem ersten Element. Wenn wir einmal den Offset zu der Basisadresse addieren, bekommen wir das zweite Element. Dementsprechend bekommen wir das dritte Element, wenn wir zwei Mal den Offset zu der Basisadresse addieren.)

Zur Veranschaulichung ist in der folgenden Abbildung ein Array dargestellt, wie es im Speicher aussehen würde.

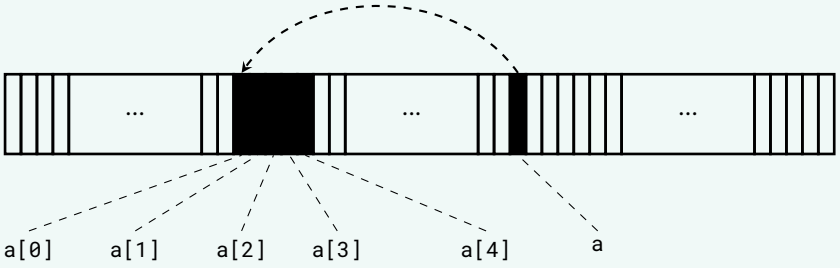

Abbildung 1.38: Abstrakte Visualisierung des Speicherplatzes eines Arrays a

Wie wir sehen können, verweist a auf die Basisadresse, also die Adresse des ersten Elements im Array.

1.7 Enums

Enum ist die Kurzform von Ennumeration, was auf deutsch Aufzählung heißt. Enum sind Klassen, die von der Klasse java.lang.Enum abgeleitet sind, und auch Referenztypen. Die Klasse java.lang.Enum bietet die Möglichkeit, vordefinierte Konstanten für Variablen festzulegen. Das ist besonders sinnvoll, wenn die Variablen eine bestimmte und feste Anzahl an Zuständen haben.

Wir wollen uns als Beispiel für Enums die Jahreszeiten anschauen, da es nur vier Jahreszeiten gibt, die alle einen festen und vordefinierten Namen haben. Wir definieren also ein Enum namens `Season`:

```
1 public enum Season {
2   SPRING, SUMMER, AUTUMN, WINTER
3 }
```

Abbildung 1.39: Jahreszeiten - Enum Season

Wenn wir in unserem Programm beispielsweise eine Anweisung ausführen wollen, die nur im Frühling stattfinden soll, können wir das folgende Schema verwenden um auf den entsprechenden Wert des Enums zuzugreifen:

Season.SPRING

Im Allgemeinen lautet also das Schema:

Enumname.Konstantenname

Im Speicher kann man sich Enums folgendermaßen vorstellen:

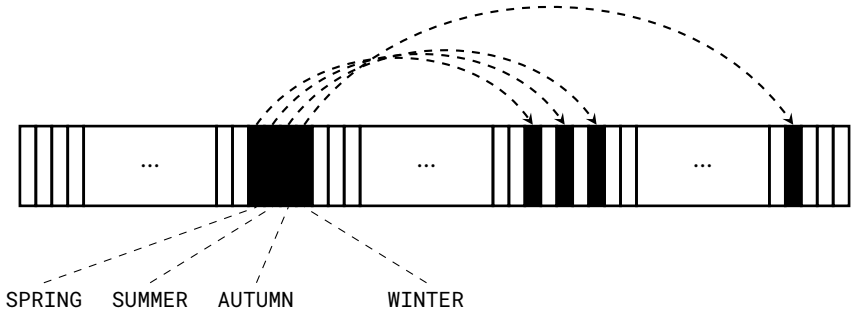

Abbildung 1.40: Abstrakte Visualisierung des Speicherplatzes des Enums Season

Wie wir sehen können, wird ein Enum intern als eine Art Array gespeichert. Wir können dann über den Namen der Konstanten auf den jeweiligen gespeicherten Wert zugreifen.

Stichwortverzeichnis

A

abstract 23, 25
Abstrakte Klasse 23
Abstrakte Klassen 25
Abstrakte Modellierung 17
Annotation 20
Array 12
Arrays 18, 32
Attribut 8, 26
Attribute 4, 6, 20
Auto 4

B

Bauanleitung 4
Bird 18
boolean 11, 28
Bücherregal 32
byte 11, 28

C

Car 4, 7, 15
Casting 22
char 11, 28
Circle 8, 9, 12, 13, 15

D

default 23
Deklarierung 10
double 11, 28
Downcasting 22
Dynamischer Typ 18

E

Einfachvererbung 25
Einfachvererbungen 17
Enum 6, 36
Enumeration 36
equals 31
Erweiterbarkeit 17
extends 18

F

Fehleranfälligkeit 17
float 11, 28
Flying 24, 25
FlyingBird 18, 26

G

Generalisierung 17
Getter 14
Globale Variablen 26

H

Hierarchie 17

I

Implementation 20
implements 23
Initialisierung 10
Instanzen 5
Instanziierung 4, 5, 20, 23, 25
int 11, 28
Interface 6, 25
Interfaces 23

J

Jahreszeiten 36

K

Klasse 4, 8, 15
Klassen 6, 17, 36
Klassenattribute 8, 9, 12, 13
Klassenkonstante 10
Klassenkonstanten 23
Klassenmethoden 12, 13
Komponententyp 32
Konstanten 26, 36
Konstruktor 15
Konventionen 8
Kreis 8

L

Lokale Variablen 11, 26
long 11, 28

M

Methoden 4, 6, 12, 20, 26, 27
Methodensignatur 13
Methodentabelle 21
Modifier 6

N

$\mathbb{N}$ 12
Natürliche Zahlen 12
new 5, 15, 28, 32
NonFlyingBird 18

O

Objekt 4, 8, 12, 13, 21
Objektattribute 8, 11, 13, 23
Objektgleichheit 30
Objektidentität 28
Objektkonstanten 11
Objektmethode 12
Objektmethoden 13, 23
Objektorientierte Programmiersprache 4
Objektorientierten Programmierung 17
Objektorientierung 17
Override 20

P

Package 6
package-private 6
Parameter 12, 13
Primitive Datentypen 28, 30
private 6, 23
protected 6
public 6, 23

R

Rectangle 19, 21, 29
Redundanz 17
Referenz 28
Referenztyp 11
Referenztypen 11, 28, 31
Rückgabetyp 12, 13

S

Schablone 23
Schnittstelle 23
Schwan 24
Scope 26
Season 36
Setter 14
short 11, 28
Speicher 21, 29–32, 35, 36
Speicheradresse 35
Spezialisierung 17
Square 19–21
Standardwerte 11
static 9
Static-Block 10
Statischer Typ 18
String 5, 11
Student 9, 11, 15
Subtypen 18
super 16
Supertyp 18
Swan 24, 25
Swimming 24

T

this 15, 27

U

Überladen von Methoden 13, 20, 26
Überschreiben von Methoden 20
UML 17
Upcasting 22

V

Variablen 26
Vererbung 17
Vertrag 23
Vogel 18
void 12

W

Wertgleichheit 28, 30

Z

Zugriffsebenen 6
Zugriffsmodifikatoren 6, 8, 26, 27
Zuweisung 10