Programmieren lernen mit Java

Eine Einführung auf Hochschulniveau

Dominikus Herzberg

23. Dezember 2024

Table of contents

Vorwort

Dieses Buchprojekt hat seinen Zündungsmoment im Wintersemester 2024/25 gefunden. Meine Studierenden aus der Veranstaltung “Programmieren 1” wünschten sich weitere Aufklärung zur Objektorientierung.

Dieses Schreibprojekt ist derzeit nicht terminiert. In den Git-Repositories zu meinen Programmierveranstaltungen haben sich sehr viel Material und Codebeispiele angesammelt. Das will nun neu sortiert, zugeordnet und überarbeitet werden.

Ich bin über alle Korrekturen und entdeckte Fehler dankbar.

Dominikus Herzberg https://www.thm.de/mni/dominikus-herzberg

1 Einfache Klassen

Eine Klasse definiert ein Kodierschema für seine Instanzen (damit sind die Variablen, genauer Instanzvariablen gemeint) und Umgangsweisen mit dem Kodierungsschema (Methoden, genauer Instanzmethoden).

In der Objektorientierung strebt man an, die Realisierung der Implementierung, d.h. die Details der Kodierung (die Variablen) und die Details des Umgangs mit der Kodierung (die Methoden), zu verbergen. Das geschieht in aller Regel dadurch, dass die intern geführten Variablen *private* gesetzt werden und dass die Methoden keinen Einblick in den zugrundeliegenden Programmcode geben.

Das heißt, anders gesagt: Mit Klassen werden zusammengesetzte Datentypen (im Javasprech "Referenztypen") erstellt, wobei die Methoden die entscheidenden Abstraktionsebene sind, um mit den Instanzen einer Klasse zu arbeiten.

> Aus pragmatischen Gründen werden wir bei der Arbeit mit der JShell Variablen nicht *private* setzen, damit wir während der Entwicklungsarbeit sehen können, welche Datenwerte in den Variablen gespeichert sind. Ein anderer Weg ist, über die `toString`-Methode Einblicke in maßgebliche Variablenwerte zu geben.

1.1 Das Schema einer Klassendeklaration

Das Schema einer einfachen Klassendeklaration sieht wie folgt aus:[1]

```
class <Name> {
    <Deklaration von Klassenvariablen mit `static`>
    <Deklaration von Instanzvariablen>
    <Deklaration von Klassenmethoden mit `static`>
    <Deklaration von Instanzmethoden>
}
```

Die Reihenfolge der Deklarationen von Variablen und Methoden spielt keinerlei Rolle und kann beliebig sein.

Variablen werden wie gewohnt deklariert nach dem Schema

```
<Typ> <VariablenName>;
```

Dazu kommen kann die Zuweisung eines Initialwertes

```
<Typ> <VariablenName> = <InitExpression>;
```

Bei Klassenvariablen wird ein `static` vorangestellt, z.B.:

```
    static initCounter = 0;
```

Methoden werden nach diesem Schema deklariert; bei Klassenmethoden wird ein `static` vorangestellt

[1] Ich unterschlage hier, dass in dem Schema zwei Arten von Initialisierungsblöcken vorkommen: der Block `static { ... }`, der zur Ladezeit der Klasse ausgeführt wird, und der Block `{ ... }`, der beim Instanziieren von Objekten ausgeführt wird.

```
<Rückgabetyp> <Methodenname>(<Parameter>) {
    <Anweisungen>
}
```

Den Parametern im Kopf einer Methode entsprechen Variablen-Deklarationen ohne initiale Zuweisung. Die Parameter werden durch ein Komma getrennt. Eine Besonderheit ist die Notation für Parameter als sogenannte Varargs. Varargs sammeln beim Aufruf der Methode beliebig viele Argumentwerte des angegebenen Typs als Array (`<Typ>[] ParameterName`) auf.

```
<Typ>... <ParameterName>
```

1.2 Beispiel: Klasse `Person` mit Instanzvariablen

Sobald eine Klasse deklariert und verbucht ist, können Instanzen von ihr mithilfe des `new`-Operators erzeugt werden. Mangels geeigneter zeichnerischer Möglichkeiten (Sie erinnern sich, Objekte sind "Kreise"), notieren wir den Verweis auf ein Objekt, das stets eine Instanz einer Klasse ist, wie folgt:

```
Objekt := <Klassenname>@<Identifier>{ <Variablenwerte> }
```

- Der Klassenname entspricht dem Namen der Klasse, die beim `new`-Operator angegeben ist
- Der Identifier ist eine eindeutige Zahl, die sich von den Identifiern aller anderen Objekte unterscheidet.
- Der jeweilige Variablenwert verweist mit einem Pfeil `->` auf Referenztypen. Ist der Datenwert ein primitiver Typ, wird ein `=` verwendet.
- Die Darstellung der Objekte für Zeichenketten (*strings*) wird verkürzt durch die Darstellung des Literals.

Wenn Variablen ohne Zuweisung deklariert sind, werden sie im instanziierten Objekt auf die Default-Werte gesetzt, d.h. `false` bei `boolean`, Null bei Zahlenwerten und `null` bei Referenztypen.

```
class Person {
    int age;
    String name;
}

Person p1 = new Person(); // p1 -> Person@1{ age = 0, name -> null }
p1.name = "Ada";          // p1 -> Person@2{ age = 0, name -> "Ada" }

Person p2 = new Person(); // p2 -> Person@2{ age = 0, name -> null }
p2.name = "Mo";           // p2 -> Person@2{ age = 0, name -> "Mo" }
p2.age = 20;              // p2 -> Person@2{ age = 20, name -> "Mo" }
```

Wie man sieht, kann man mittels des Punkt-Operators (`.`) vom Objekt aus auf die Variablen über die Variablenname zugreifen. Links vom Zuweisungs-Operator (`=`) entspricht dem Zugriff eine Schreiboperation, rechts davon eine Leseoperation.

Was man hier lernt:

- Die Klasse `Person` hat genau zwei Instanzvariablen, `age` und `name`, die mit ihren Defaultwerten (Grundwerten) initialisiert werden, `0` bzw. `null`.
- Der `new`-Operator ruft den Default-Konstruktor auf. Dieser Konstruktor wird nur dann erstellt, wenn man keine Konstruktoren in der Klasse deklariert.

Der unsichtbare Code des Default-Konstruktors sieht wie folgt

```
Person() {
    super();
}
```

Der Konstruktor ruft schlicht den Konstruktor der Oberklasse auf. Jede Klasse hat eine Oberklasse. Wenn die Oberklasse nicht explizit mit `extends` angegeben wird, ist es die Klasse `Object`.

- Nachdem eine Instanz erzeugt ist, kann man die Werte der Instanzvariablen auslesen bzw. ihnen neue Werte zuweisen.

Ausprobieren:

Ergänzen Sie eine `toString`-Methode, so dass man `age` und `name` zu einer Instanz angezeigt bekommt.

1.3 Beispiel: `Person` mit Konstruktoren und Methoden

Da das Alter (*age*) einer Person etwas ist, was sich mit der Zeit verändert, ist es keine so gute Idee, das Alter als Variablenwert zu modellieren. Wir überarbeiten die Klasse `Person`.

1.3.1 Hilfsklassen für Datumsangaben

Für die Arbeit mit Datumsangaben stellt das JDK die `java.time`-Bibliothek zur Verfügung. Für unsere Zwecke benötigen wir die Klasse `LocalDate` und `Period`, die wir zuvor importieren. Schauen Sie sich die Klassen an (die Klassenname oben sind auf die Oracle-Dokumentation verlinkt) und lesen Sie nach, wie man eine Instanz von `LocalDate` erzeugt und wie man aus zwei Datumsangaben einen Zeitraum `Period` berechnet.

Im JDK gibt es viele Klassen für Standardaufgaben, deren Programmierung ausgereift ist. Zum Beispiel ist der Umgang mit Datumsangaben nicht trivial (Zeitzonen, Schaltjahre etc.). Man sucht sich heraus, was man braucht und verwendet die angebotenen Klassen mit ihren Methoden. Mit der JShell kann man rasch den Gebrauch solcher Klassen ausprobieren.

```
jshell> import java.time.LocalDate;

jshell> LocalDate.of(2024,12,16)
$3 ==> 2024-12-16

jshell> import java.time.Period;

jshell> Period.between($3, LocalDate.of(2026,12,16))
$5 ==> P2Y

jshell> $5.getYears()
$6 ==> 2
```

Was man hier lernt:

- Die `toString`-Methode ist in beiden Klassen angepasst, damit man weiß, wofür das Objekt steht. `P2Y` ist also eine Kurzdarstellung einer Periode von 2 Jahren (*years*).
- Die Instanzen dieser beiden Klassen bieten einige Methoden an. Hier verwenden wir die `getYears`-Methode.
- `LocalDate.of` und `Period.between` sind Klassenmethoden (*static methods*). Woran man das erkennt? `LocalDate` und `Period` sind Klassen. Wären `of` und `between` Instanzmethoden, dann hätte man zuvor mit `new` ein Objekt anlegen müssen. Interessanterweise wird dennoch eine Instanz der entsprechenden Klasse zurückgeliefert. Wie das geht, schauen wir uns später bei der Klasse `Person` an.

1.3.2 Die überarbeitete Klasse `Person`

Statt einer Altersangabe geben wir im Konstruktor ein Geburtsdatum `birth` und optional (wahlweise) einen Namen `name` an.

Damit niemand Zugriff auf die Variable `birth` hat, denn das Geburtsdatum soll nicht verändert werden können (auch nicht in der JShell), ist die Variable auf `private` gesetzt. Der Name kann auch später nach der Geburt oder irgendwann im Leben einer Person angepasst werden. Der Zugriff auf die Variable ist nicht reglementiert.

Das Alter wird über die Methode `getAge` abhängig von einem gegebenen Datum (oder dem aktuellen Tagesdatum) berechnet. Für Zeiträume die *vor* der Geburt liegen, wir eine "negative" Dauer angegeben.

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

class Person {
    private LocalDate birth;
    String name;
    Person(LocalDate birth, String name) {
        this.birth = birth;
        this.name = name;
    }
    Person(LocalDate birth) {
        this(birth, "");
    }
    Period getAge(LocalDate date) {
        return Period.between(birth, date);
    }
    Period getAge() {
        return getAge(LocalDate.now());
    }
```

```
    @Override public String toString() {
        return super.toString() + "{ " +
            "birth -> " + birth + ", " +
            "name -> \"" + name + "\" }";
    }
}
```

Was man hier lernt:

- Konstruktoren sind eine besondere Form der Methode (*Initialisierungsmethoden*), die eine Instanz der Klasse zurückgeben, es aber erlauben, z.B. übergebene Initialisierungswerte vor der Zuweisung zu überprüfen (etwa mittels `assert`).
- Mit `this()` kann man einen anderen Konstruktor (mit einer anderen Signatur) aufrufen. Auf diese Weise wird hier der Default-Wert (Standardwert) für `name` auf eine leere Zeichenkette gesetzt. (Alternativ hätte man `String name = "";` deklarieren können.)
- Die Methode `getAge` ist für zwei verschiedene Signaturen deklariert. Auch hier sieht man, wie man Default-Werte setzen kann: Bei einem Aufruf ohne Argument wird das aktuelle Tagesdatum genutzt.
- Die `toString`-Methode passt die Repräsentation eines Objekts an die obige Notation an. Mit `super` bedient man sich der Standardrepräsentation der Oberklasse (stets `Object` bei Klassen, die keine Oberklasse mit `extends` angeben) und ergänzt mit `+` (String-Konkatenation, also String-Verkettung) die weiteren Angaben.
- Da die Klasse `Person` die Klasse `Object` erweitert, überschreiben wir mit einer eigenen `toString`-Methode die der Oberklasse. Unsere lokale `toString`-Methode darf die Zugriffsrechte nicht reduzieren und muss deshalb ein `public` voranstellen.
- Das `@Override` ist eine sogenannte Annotation, die den Java-Compiler dazu veranlasst zu überprüfen, ob es in der Oberklasse eine Methode dieses Namens gibt. Wenn nicht, wird der Über-

setzungsvorgang mit einer Fehlermeldung abgebrochen. (Hinweis: Annotationen wie `@Override` werden in aller Regel nicht in die Methodenzeile geschrieben, sondern in einer separaten Codezeile vor dem Methodenkopf.)

- Es ist nicht notwendig, externe Klassen, die man mit `import` zur Nutzung bereitstellt, im Detail zu kennen und zu verstehen.

  Eine `getAge`-Methode bedeutet nicht, dass es in der Klasse eine Variable namens `age` gibt – wie das Beispiel zeigt. Es handelt sich hier *nicht* um eine sogenannte "Getter-Methode".

Interaktion:

```
jshell> Person p1 = new Person(LocalDate.of(2000,6,1), "Jo")
p1 ==> Person@4a87761d{ birth -> 2000-06-01, name -> "Jo" }

jshell> p1
p1 ==> Person@4a87761d{ birth -> 2000-06-01, name -> "Jo" }

jshell> p1.name
$39 ==> "Jo"

jshell> p1.birth
|  Fehler:
|  birth hat private-Zugriff in Person
|  p1.birth
|  ^------^

jshell> p1.getAge()
$40 ==> P24Y6M15D

jshell> p1.getAge().getYears()
$41 ==> 24
```

```
jshell> p1.getAge(LocalDate.of(2010,7,1)).getYears()
$42 ==> 10

jshell> p1.getAge(LocalDate.of(2010,5,1)).getYears()
$43 ==> 9

jshell> p1.getAge(LocalDate.of(1995,5,1)).getYears()
$44 ==> -5

jshell> p1.getAge(LocalDate.of(1995,7,1)).getYears()
$45 ==> -4
```

Ausprobieren:

Was passiert, wenn man den Rumpf der `toString`-Methode folgendermaßen verkürzt?

```
    @Override public String toString() {
        return super.toString();
    }
```

Was passiert, wenn man sich bei `toString` verschreibt (`ToString`)? Vergleichen Sie es mit `@Override` und ohne die Annotation.

```
    @Override public String toString() {
        return super.toString() + "{ " +
            "birth -> " + birth + ", " +
            "name -> \"" + name + "\" }";
    }
```

1.4 Beispiel: `Person` mit statischer `of`-Methode

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

class Person {
    private LocalDate birth;
    private String name = "";

    static Person of(LocalDate birth) {
        return new Person(birth);
    }

    private Person(LocalDate birth) {
        LocalDate now = LocalDate.now();
        assert birth.isBefore(now) ||
               birth.isEqual(now) : "Birthdate must not be in the future";
        this.birth = birth;
    }
    Period getAge(LocalDate date) {
        return Period.between(birth, date);
    }
    Period getAge() {
        return getAge(LocalDate.now());
    }
    Person setName(String name) {
        assert Objects.nonNull(name) : "Name must not be null";
        name = name.trim(); // entfernt vor- und nachstehende Leerzeichen
        assert !name.isEmpty() : "Name must not be empty";
        this.name = name;
        return this;
    }
```

```
    String getName() {
        return name;
    }
    @Override public String toString() {
        return super.toString() + "{ " +
            "birth -> " + birth + ", " +
            "name -> \"" + name + "\" }";
    }
}
```

Was man hier lernt:

- Nun sind die Variablen `name` und `birth` außerhalb der Klasse nicht einsehbar. Für `name` sind zwei Methoden notwendig geworden: `setName` und `getName`. Allerdings, und das ist der Vorteil, wenn man Variablenwerte über den Umweg von Methoden verändert, geschieht nun eine Überprüfung, ob der Name gewissen Anforderungen genügt.
- Der Konstruktor ist aufgrund von `private` nicht mehr von außerhalb der Klasse aufrufbar. Man *muss* jetzt Instanzen über die Klassenmethode `of` erzeugen.
- Die Methode `setName` ist nicht `void` (was man auch hätte machen können), sondern sie gibt die eigene `Person`-Instanz mittels `return this;` zurück. Das erlaubt es, Aufrufe zu verketten, siehe unten bei der Initialisierung von `p2`.
- So kurz der Code auch ist, er implementiert eine eigene Logik des Umgangs mit der Idee einer "Person" und kapselt sich sauber von der "Umwelt" ab. Eine Person kann ohne Namen erzeugt werden, allerdings darf das Geburtsdatum nicht in der Zukunft liegen. Ein später nachgelieferter oder auch später geänderter Name kann niemals "leer" sein, er muss aus mindestens einem Buchstaben bestehen. (Selbstredend kann man auch andere Datenlogiken als

sinnvoll erachten. Das hängt vom Einsatzzweck und dem Kontext ab.)

Interaktion:

```
jshell> Person p1 = Person.of(LocalDate.of(2000,12,6))
p1 ==> Person@3224f60b{ birth -> 2000-12-06, name -> "" }

jshell> p1.setName("Niki")
$10 ==> Person@3224f60b{ birth -> 2000-12-06, name -> "Niki" }

jshell> p1.getName()
$11 ==> "Niki"

jshell> p1.getAge(LocalDate.of(2024,12,12))
$12 ==> P24Y6D

jshell> Person p2 = Person.of(LocalDate.of(2000,12,6)).setName("Ty")
p2 ==> Person@614ddd49{ birth -> 2000-12-06, name -> "Ty" }
```

Fragen:

Hätte man für die Zusicherung in `setName` auch `assert name != null` verwenden können?

Ja, selbstverständlich. Gewöhnen Sie sich dennoch das sprechende `Objects.nonNull(...)` an. Die Klasse `Objects` bietet noch ein paar andere hilfreiche Methoden an.

Warum beginnt der Konstruktor mit `LocalDate now = LocalDate.now();`? Man hätte die Zeile auch weglassen können und `birth.isBefore(LocalDate.now())` und `birth.isEqual(LocalDate.now())` schreiben können. Oder?

Es ist ein äußerst seltener Sonderfall: Zwischen den beiden `LocalDate.now()` könnte die Tagesgrenze liegen. Das würde zwar in diesem Fall keinen

Schaden anrichten, aber es ist besser, mit der Variable `now` einen definierten Tageszeitpunkt zu wählen, gegenüber dem die beiden Vergleiche vorgenommen werden.

Ausprobieren:

Verändern Sie die Methode `setName` auf den Rückgabetyp `void`. Was müssen Sie am Code ändern? Was ändert sich an der Interaktion?

1.5 `Counter`-Beispiel: Klassen- und Instanzmethoden bzw. -Variablen

In einer Klassendeklaration sind Klassenvariablen von Instanzvariablen durch das Schlüsselwort `static` unterscheidbar; gleiches gilt für Klassenmethoden.

```
class Counter {
    static int number  = 0;
    int tick = 0;
    static String showClass() {
        return "Counter{ counter = " + number + " }";
    }
    Counter() {
        number++;
    }
    Counter click() {
        tick++;
        return this;
    }
    @Override public String toString() {
        return super.toString() + "{ tick = " + tick + " }";
    }
}
```

Was man hier lernt:

- Klassenvariablen und -methoden (`static`) gelten nur im Kontext der Klasse. Darum müssen Sie von außerhalb (z.B. von der JShell) über den Klassennamen adressiert werden. In dem Fall also `Counter.number` bzw. `Counter.show()`. Innerhalb der Klasse ist die Erwähnung der Klasse nicht nötig.
- Instanzvariablen und -methoden gelten nur für die Instanz. Um sich die Instanz zu erhalten, muss sie in einer Variablen gespeichert sein, z.B. in `Counter c1`. Dann erfolgt der Zugriff auf die Variable bzw. die Methode über `c1.tick` bzw. `c1.click()`.
- Innerhalb der Klasse kann eine Instanz auch über das Schlüsselwort `this` auf die eigene Variable bzw. die Methode zugreifen: `this.tick` bzw. `this.click()`. Allerdings verzichtet man in aller Regel auf `this`, der Compiler fügt das automatisch hinzu.

Interaktion:

```
jshell> Counter.number
$3 ==> 0

jshell> Counter c1 = new Counter()
c1 ==> Counter@2f686d1f{ tick = 0 }

jshell> Counter.number
$5 ==> 1

jshell> Counter c2 = new Counter()
c2 ==> Counter@6e06451e{ tick = 0 }

jshell> Counter.number
$7 ==> 2

jshell> c1.click().click().click()
```

```
$8 ==> Counter@2f686d1f{ tick = 3 }

jshell> c2.click()
$9 ==> Counter@6e06451e{ tick = 1 }

jshell> Counter.number
$10 ==> 2

jshell> c1.tick
$11 ==> 3

jshell> c2.tick
$12 ==> 1
```

> Mit Klassenvariablen und Klassenmethoden sollte man nur arbeiten, wenn der Gebrauch gut begründet ist. In der Mehrzahl der Fälle möchte man den Kontext von Instanzen nutzen, damit man von den Vorteilen der Objektorientierung profitiert.

1.6 Ausblick: `record` (Datenklasse)

In modernem Java würde man die Klasse `Person` eher als Datenklasse (`record`) umsetzen. Der Code fällt dann kürzer aus.

> Zu Datenklassen gibt es vieles mehr zu sagen. Dieser Abschnitt soll nur zeigen, wie eine Umsetzung als Datenklasse aussieht und einer ersten Berührung mit Datenklassen dienen.

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

record Person(LocalDate birth, String name) {
```

```
    public Person {
        assert Objects.nonNull(name) : "Name must not be null";
        name = name.trim(); // entfernt vor- und nachstehende Leerzeichen
        assert !name.isEmpty() : "Name must not be empty";
    }
    Person(LocalDate birth) {
        this(birth, "");
    }
    Person setName(String name) {
        return new Person(birth, name);
    }
    Period getAge(LocalDate date) {
        return Period.between(birth, date);
    }
    Period getAge() {
        return getAge(LocalDate.now());
    }
}
```

- Die im Konstruktor angegebenen Variablen sind (1) `final`, d.h. unveränderlich (*immutable*) und es werden (2) für die Variablen automatisch Abrufmethoden angelegt, hier `birth()` und `name()`.
- Aufgrund der Immutabilität (nicht Überschreibbarkeit) von Variablen in Datenklassen, kann man auch keine Variablenwerte mehr nach dem Konstruktoraufruf ändern! Wenn man also den Namen mit `setName` anpassen möchte, *muss* man eine neue Instanz von `Person` zurückgeben, die das Geburtsdatum beibehält, aber mit einem neuen Namen versehen ist. Datenklassen verlangen nach einem anderen Programmierstil!
- Die Überprüfung für den Wert von `name` muss in den Konstruktor wandern. Grund ist wieder die Immutabilität.
- Datenklassen erzeugen automatisch eine `toString`-Methode, die die Variablenwerte aus dem Konstruktur enthält und anzeigt. Das ist in

den meisten Fällen praktisch und hilfreich.
- Wenn man möchte, könnte man auch hier eine Klassenmethode namens `of` anlegen, was bei Datenklassen aber in der Regel weniger Sinn macht.

Interaktion:

```
jshell> new Person(LocalDate.of(2000,1,1), "Sy")
$25 ==> Person[birth=2000-01-01, name=Sy]

jshell> Person p = new Person(LocalDate.of(2000,1,1), "Sy")
p ==> Person[birth=2000-01-01, name=Sy]

jshell> p.name()
$27 ==> "Sy"

jshell> p.birth()
$28 ==> 2000-01-01

jshell> p = p.setName("Zu")
p ==> Person[birth=2000-01-01, name=Zu]

jshell> p.getAge()
$30 ==> P24Y11M15D
```

Fragen:

Warum heißt es in der Interaktion `p = p.setName("Zu")`?

Wenn `p` eine Variable ist, die ein und die selbe Person repräsentieren soll, dann muss die neue Instanz von `Person` dem `p` zugewiesen werden. Grundsätzlich sollte die neue Instanz einer Variablen zugewiesen werden, weil ein einfaches `p.setName("Zu")` ein nicht gespeichertes Objekt zur Folge hätte, was somit nicht mehr zugreifbar ist. In der JShell erfolgt die Zuweisung zu einer Variablen automatisch, damit man leichter experimentieren kann.

1.7 Beispiel `StringComposer`: Methodenüberladung (*method overloading*)

Die Signatur einer Methode setzt sich zusammen aus dem Methodennamen und den Typen ihrer Parameter. Signaturen unterscheiden sich im Methodennamen und, bei gleichem Methodennamen, in der Anzahl und Reihenfolge der Typparameter. Ausdrücklich *nicht* zur Signatur zählen der Rückgabetyp einer Methode, Ausnahmen (die kann man im Methodenkopf angeben) und die Parameternamen.

Gibt es zu einem Methodennamen mehrere unterschiedliche Signaturen, spricht man von Methodenüberladung. Das meint: Man kann die Methode unter ihrem Namen mit unterschiedlichen Argumenttypen und/oder einer unterschiedlichen Anzahl von Argumenten aufrufen.

Das nachstehende Beispiel zeigt die Methodenüberladung für die Methode `append`. Die Signaturen unterscheiden sich eindeutig (sonst würde der Compiler einen Fehler melden), so dass beim Methodenaufruf stets die passende Methode ausgewählt wird.

```
class StringComposer {
    String s;
    StringComposer(String s) { this.s = s; }
    String append(int n) { return s += n; }
    String append(String text) { return s += text; }
    String append(String text, int times) {
        for (int i = 1; i <= times; ++i) s += text;
        return s;
    }
    String append(int n, int times) { return append("" + n, times); }
    String append(String... texts) {
        for (String text : texts) s += text;
        return s;
    }
```

```
    String append(String a, String b) {
        return s += a + b;
    }
    @Override public String toString() {
        return super.toString() + "{ s -> \"" + s + "\" }";
    }
}
```

Was man hier lernt

- Die Signaturen lauten `append(int)`, `append(String)`, `append(String, int)` usw. Sie müssen alle unterschiedlich sein. Man sagt: Die Methode `append` ist überladen. (Im Beispiel sind alle Rückgabetypen gleich. Das müsste nicht so sein.)
- Beim Aufruf wird die spezifischste Methode gewählt. Bei einer bzw. zwei Zeichenketten im Aufruf, werden die Methoden mit den Signaturen `append(String)` bzw. `append(String, String)` gewählt. Bei keiner, drei oder mehr Zeichenketten greift `append(String...)`.
- Eine Zuweisung wie z.B. `s += text` verändert nicht nur eine Variable, sie gibt zudem den Zuweisungswert zurück. Das erlaubt neben der Zuweisung eine Rückgabe des Werts mittels `return`.

Interaktion

```
jshell> StringComposer sc = new StringComposer("Hi")
sc ==> StringComposer@6500df86{ s -> "Hi" }

jshell> sc.append(3)
$3 ==> "Hi3"

jshell> sc.append("You")
$4 ==> "Hi3You"
```

```
jshell> sc.append("c", 3)
$5 ==> "Hi3Youccc"

jshell> sc.append(5, 3)
$6 ==> "Hi3Youccc555"

jshell> sc.append("a", "b")
$7 ==> "Hi3Youccc555ab"

jshell> sc.append("a", "b", "c", "d")
$8 ==> "Hi3Youccc555ababcd"

jshell> sc.append()
$9 ==> "Hi3Youccc555ababcd"

jshell> sc
sc ==> StringComposer@6500df86{ s -> "Hi3Youccc555ababcd" }
```

Ausprobieren

Verändern Sie den Rückgabewert einer `append`-Methode in `void`. Ändert das etwas an der Methodenüberladung?

Sind Sie sicher, dass `sc.append("a", "b")` nicht die Methoden mit der Signatur `append(String...)` aufruft? Verändern Sie den Code in den Methodenrümpfen so, dass unterscheidbar ist, welche Methode aufgerufen wird. (Im Zweifel dürfen Sie ausnahmsweise `System.out.println` verwenden.)

1.8 `Counter`: Initiale Codeblöcke `static { ... }` und `{ ... }`

Das obige Schema für eine Klassendeklaration hat zwei initiale Codeblöcke unterschlagen, die in einer Klassendeklaration vorkommen können.

```
class <Name> {
    static { /* Klasseninitialisierung */ }
    { /* Instanzinitialisierung */ }
    <Deklaration von Klassenvariablen mit `static`>
    <Deklaration von Instanzvariablen>
    <Deklaration von Klassenmethoden mit `static`>
    <Deklaration von Instanzmethoden>
}
```

Man kann in einer Klassendeklaration einen statischen Block `static { ... }` einfügen, der zur Ladezeit der Klasse ausgeführt wird, und einen Block `{ ... }`, der beim Instanziieren von Objekten zur Ausführung kommt.

Nehmen wir die Klasse `Counter` und fügen in den Initialisierungblöcken `println`-Anweisungen ein zur Nachvollziehbarkeit des Zeitpunkts der Ausführung des Blocks.

```java
class Counter {
    static int number  = 0;
    int tick = 0;
    static {
        System.out.println("Executed at class loading time");
    }
    {
        System.out.println("Executed prior to constructor call");
    }
```

```
    static String showClass() {
        return "Counter{ counter = " + number + " }";
    }
    Counter() {
        number++;
    }
    Counter click() {
        tick++;
        return this;
    }
    @Override public String toString() {
        return super.toString() + "{ tick = " + tick + " }";
    }
}
```

Kopiert man den Code in die JShell und erzeugt eine erste Instanz, sieht man deutlich den Effekt der Blöcke. Der statische Init-Block (Klasseninitialisierung) wird nur ein einziges Mal aufgerufen, der andere Init-Block (Instanzinitialisierung) wird bei jeder Instanziierung aufgerufen und zwar vor der Abarbeitung des Codes im Konstuktor.

Interaktion

```
jshell> new Counter()
Executed at class loading time
Executed prior to constructor call
$2 ==> Counter@2f686d1f{ tick = 0 }

jshell> new Counter()
Executed prior to constructor call
$3 ==> Counter@6e06451e{ tick = 0 }

jshell> new Counter()
```

```
Executed prior to constructor call
$4 ==> Counter@6e1567f1{ tick = 0 }
```

Man sollte den Block für die Instanzinitialisierung nicht als Ersatz für einen Konstruktor nehmen und auf seinen Gebrauch verzichten. Gleiches gilt für den Block zur Klasseninitialisierung. Man sollte also wissen, dass es diese Blöcke gibt, sie aber nur dann einsetzen, wenn es sehr gute Gründe dafür gibt.

1.9 "Gratis"-Methoden von `Object`

Klassen, die ohne ein explizites `extends` im Kopf der Deklaration auskommen, haben ein implizites `extends Object` da stehen. Die Klassen dieses Kapitels kann man also gedanklich im Deklarationskopf lesen als:

```
class Person extends Object { /* body */ }

class Counter extends Object { /* body */ }

class StringComposer extends Object { /* body */ }
```

Jede "einfache" Klasse erweitert die Klasse `Object`. Das hat zur Folge, dass jede Klasse die folgenden Methoden von `Object` übernimmt oder ausdrücklich mit einer eigenen Implementierung überschreibt:

`protected Object clone()` Erzeugt eine Kopie des Objekts

`public boolean equals(Object other)` Objekte vergleicht man in aller Regel nicht mit dem Gleichheitsoperator `==` (es sei denn, man möchte die Identität, d.h. die Referenzgleichheit der Objekte überprüfen), sondern mit Hilfe der `equals`-Methode, die es erlaubt, die inhaltliche Gleichheit von Objekten anhand ihrer Datenwerte zu überprüfen. Allerdings muss man diese Methode überschreiben und mit einer

passenden Implementierung ersetzen. Mehr dazu in einem gesonderten Abschnitt. Die Standard-Implementierung in `Object` sieht wie folgt aus:

```
public boolean equals(Object other) {
    return this == other;
}
```

`public final Class<?> getClass()` Liefert die Laufzeit-Repräsentation der Klasse des Objekts.

`public int hashCode()` Gibt den HashCode eines Objekts zurück. Der HashCode für ein Objekt wird benötigt für Datenstrukturen wie z.B. die `HashMap`. Zwei Objekte die gleich sind, müssen den gleichen HashCode haben.

`public String toString()` Liefert eine Repräsentation, d.h. eine textuelle Selbstdarstellung des Objekts zurück. Die Standards-Implementierung von `Object` sieht (geringfügig vereinfacht) wie folgt aus:

```
public String toString() {
    return getClass().getName() + "@" +
           Integer.toHexString(hashCode());
}
```

Die noch verbleibenden Methoden `notify()`, `notifyAll()` und die überladene Methode `wait(...)` sind relevant für die nebenläufige Programmierung mit Threads. Das wollen wir an dieser Stelle vorerst übergehen.

1.10 Was genau ist ein Objekt?

Technisch gesprochen belegt ein Objekt einen Speicherbereich in einem Speicher, der sich Heap (engl. für “Haufen”) nennt. Der Heap ist ein

verwalteter Speicherbereich. Die Speicherverwaltung schafft Platz für neu dazukommende Objekte, reorganisiert gelegentlich den Speicher (z.B. wenn es zu viele Lücken gibt, so dass neue Objekte Platz finden) und kümmert sich darum, dass Objekte, die direkt oder indirekt durch die aktiven Referenzen nicht mehr erreicht werden, aus dem Heap entfernt werden; dies nennt sich Garbage Collection (GC, Speicherbereinigung).

Wenn in Java der `new`-Operator zum Einsatz kommt, wird ein Objekt erzeugt. Nehmen wir den obigen Code zur Klasse `Person` zur Anschauung (die Variante mit der `of`-Methode):

```
jshell> Person p = Person.of(LocalDate.of(2000,1,1)).setName("Jes")
p ==> Person@5ce81285{ birth -> 2000-01-01, name -> "Jes" }
```

Zum Anlegen des Objektes werden in der Klasse `Person` lediglich die Instanzvariablen (die "normalen" und nicht als `static` ausgewiesenen Variablendeklarationen) betrachtet. Entsprechend der dortigen Typangaben wird der benötigte Speicherplatz für die Variablen bereitgestellt. Auf dem Heap kommt das Objekt mit folgendem Speicheraufbau unter einer dynamisch vergebenen Speicheradresse zum Tragen:

Table 1.1: Speicherorganisation eines Objekts, das eine Instanz von `Person` ist

Offset	Bytes	Feld	Beschreibung
0	8	Mark Word	Metadaten: HashCode, Synchronisation, GC
8	8	Class Pointer	Verweis auf die Klassenbeschreibung
16	4	birth	Referenz auf `LocalDate`-Instanz
20	4	name	Referenz auf `String`-Instanz
24	8	Padding	Auffüll-Bytes für Gesamtgröße von 32 Bytes

Ohne hier auf alle Details einzugehen: Neben der Speicheradresse (dem entspricht die Referenz) für diesen Speicherblock, weist die Tabelle die

entscheidenden und wichtigen Infos aus: Den HashCode (sofern er nicht über eine überschriebene `hashCode`-Methode berechnet wird), den Verweis auf die Klasse, von der das Objekt instanziiert wird (erhältlich über `getClass()`), und die Speicherbedarfe für die Instanzvariablen, hier Referenzen auf die Objekte von den Klassen `LocalDate` und `String`.

Die Variable `p` im obigen Beispiel enthält also die Referenz auf ein Objekt, das eine Instanz der Klasse `Person` ist. Der Referenz entspricht die Speicheradresse, an der der Speicherblock für das Objekt beginnt. Die Speicheradresse ist nicht einsehbar, sie wird aber intern herangezogen, wenn zwei Objekte auf Gleichheit mit dem Operator `==` geprüft werden.

Als Mensch helfen die technischen Einblicke für das Verständnis, aber Bilder sind besser, um einen Gesamtblick zu bekommen. Man kann Referenzen sehr gut als Pfeile (= Verweise) auf Objekte verstehen, wobei der Name am Pfeil den Variablennamen kennzeichnet. Objekte sind in dem nachstehenden Bild als Kästen mit abgerundeten Ecken dargestellt. In dem Objekt steht entweder der Name der Klasse, von der das Objekt instanziiert wurde, oder seine Repräsentation, wie sie durch die `toString`-Methode bestimmt ist.

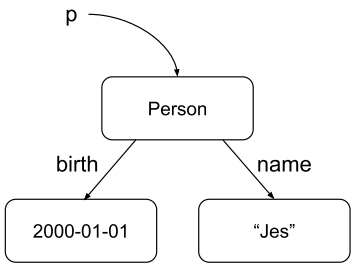

Figure 1.1: Eine Instanz von Person