HANSER

Leseprobe

Dietmar Ratz, Jens Scheffler, Detlef Seese, Jan Wiesenberger

Grundkurs Programmieren in Java

ISBN: 978-3-446-42663-4

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-42663-4

sowie im Buchhandel.

Kapitel 8

Der grundlegende Umgang mit Klassen

Im letzten Kapitel haben wir erfahren, dass sich die objektorientierte Philosophie aus den vier Konzepten Generalisierung, Vererbung, Kapselung und Polymorphismus zusammensetzt. Wir haben jeden dieser Begriffe – in der Theorie – erklärt und uns die Idee klar zu machen versucht, die hinter der Objektorientierung steht. Wir haben jedoch noch nicht gelernt, diese Konzepte in Java umzusetzen.
In diesem und dem folgenden Kapitel soll dieser Mangel behoben werden. Anhand einfacher Beispiele werden wir lernen, wie sich Klassen auch in Java zu mehr als nur einfachen Datenspeichern mausern.

8.1 Vom Referenzdatentyp zur Objektorientierung

In diesem Kapitel werden wir versuchen, verschiedene Aspekte im Leben eines *Studierenden* zu modellieren. Wir beginnen hierbei mit einer einfachen Klasse, wie wir sie schon aus den vorigen Kapiteln kennen:

```
1 /** Diese Klasse simuliert einen Studenten */
2 public class Student {
3
4   /** Der Name des Studenten */
5   public String name;
6
7   /** Die Matrikelnummer des Studenten */
8   public int nummer;
9 }
```

Wie Sie sehen, haben wir die Klasse allerdings nicht `Studierender` genannt, was dem aktuellen geschlechtsneutralen Sprachgebrauch an den Hochschulen eher entsprechen würde. Der Einfachheit (und Kürze) halber haben wir uns dazu ent-

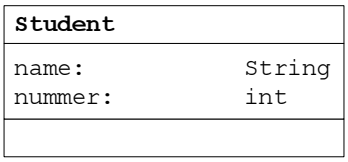

Abbildung 8.1: Die Klasse `Student`, erste Version

schlossen, die Klasse `Student` zu nennen. Natürlich soll diese Klasse aber sowohl weibliche als auch männliche Student(inn)en modellieren.[1]
Abbildung 8.1 zeigt diesen einfachen Klassenaufbau im UML-Klassendiagramm. Unsere Klasse setzt sich aus zwei Instanzvariablen namens `name` und `nummer` zusammen. Erstgenannte speichert den Namen des Studierenden, Letztere die Matrikelnummer.[2] Wir können diese Klasse nun wie gewohnt instantiieren (d. h. Objekte aus ihr erzeugen) und diese dann mit Werten belegen:

```
Student studi = new Student();
studi.name = "Karla Karlsson";
studi.nummer = 12345;
```

Bis zu diesem Punkt haben wir an unserer Klasse keine Arbeiten vorgenommen, die wir nicht aus Kapitel 5 schon zu Genüge kennen. Wir wollen diesen Entwurf nun bezüglich unserer vier Grundprinzipien überprüfen:

- Bei unserer Klasse `Student` handelt es sich um eine einzelne Klasse, nicht um eine Hierarchie. Wir haben somit keine weiteren Klassen und können damit keine Eigenschaften in Superklassen auslagern. Das Thema Generalisierung ist also in diesem Beispiel nicht weiter wichtig.
- Ähnliches gilt für die Bereiche Vererbung und Polymorphismus. Beide Begriffe spielen erst bei der Arbeit mit mehr als einer Klasse eine wichtige Rolle. Hiermit beschäftigen wir uns aber erst im nächsten Kapitel näher.
- Bleibt also die Frage, ob wir uns bezüglich der Kapselung für ein gutes Modell entschieden haben. Haben wir die interne Struktur unserer Klasse von der Schnittstelle nach außen getrennt? Könnten wir die Instanzvariablen einfach verändern, ohne hiermit Probleme zu verursachen?

An dieser Stelle müssen wir den letzten Punkt leider klar und deutlich verneinen. Unsere Instanzvariablen sind von außen her überall zugänglich. Wir schreiben unsere Werte direkt in sie hinein und lesen sie aus ihnen direkt wieder aus. Wenn wir die Matrikelnummer später in einem `String` ablegen wollen (z. B. weil wir eine Datenbank benutzen, die keine einfachen Datentypen versteht), müssen wir sämtliche Programme überarbeiten, die diese Variablen benutzen. Wir werden deshalb

[1] Wir hoffen, dass unsere *Leserinnen* aufgrund dieser Namenswahl das Buch jetzt nicht empört aus der Hand legen. Wir werden in Übungsaufgabe 8.2 dafür sorgen, dass man sogar explizit zwischen weiblichen und männlichen Studierenden unterscheiden kann.

[2] Eine von der Verwaltung der Hochschule vergebene eindeutige Nummer, unter der die Daten eines Studierenden hinterlegt werden.

im nächsten Abschnitt erfahren, wie wir mit Hilfe so genannter **Zugriffsmethoden** eine bessere Form der Datenkapselung erreichen.

8.2 Instanzmethoden

8.2.1 Zugriffsrechte

Wir beginnen damit, unsere Daten vor der Außenwelt zu „verstecken." Gemäß der Idee des **data hiding** sorgen wir dafür, dass niemand außerhalb der Klasse auf unsere Instanzvariablen zugreifen kann.
Um dieses Ziel zu erreichen, ändern wir die so genannten **Zugriffsrechte** für die einzelnen Variablen. Momentan haben unsere Variablen die Zugriffsrechte **`public`**, das heißt, sie sind *öffentlich zugänglich*. Konkret bedeutet es, dass jede andere Klasse auf die Variablen lesenden und schreibenden Zugriff hat. Genau das wollen wir jedoch verhindern!
Um dieses Ziel zu erreichen, setzen wir die Zugriffsrechte von **`public`** auf **`private`**. Privater Zugriff ist das genaue Gegenteil von öffentlichem Zugriff: während bei Ersterem *jede* Klasse auf die Variablen Zugriff hat, kann nun *keine* Klasse mehr auf die Variablen zugreifen, nicht einmal eigene Subklassen. Eine Ausnahme stellt natürlich eben jene Klasse dar, in der die Instanzvariablen definiert sind. Es handelt sich hierbei also wirklich um ihre *privaten* Variablen, die nur der Klasse selbst „gehören".
Abbildung 8.2 zeigt diese Modifikation im UML-Diagramm. Wir sehen, dass private Variablen durch ein Minuszeichen vor dem Variablennamen markiert werden. Fehlt dieses Symbol oder ist es durch ein Pluszeichen ersetzt, geht man von öffentlichen Zugangsrechten aus.[3]

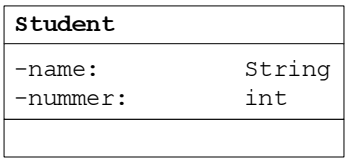

Abbildung 8.2: Die Klasse `Student`, zweite Version

Die entsprechende Umsetzung in unserem Java-Programm ist relativ einfach: Wir ersetzen lediglich das Schlüsselwort **`public`** bei den entsprechenden Variablen durch das Schlüsselwort **`private`**:

```
1  /** Diese Klasse simuliert einen Studenten */
2  public class Student {
3
4    /** Der Name des Studenten */
```

[3] Neben öffentlichem und privatem Zugriff gibt es zwei weitere Formen des Zugriffs (siehe Abschnitt 9.8.2).

```
5     private String name;
6
7     /** Die Matrikelnummer des Studenten */
8     private int nummer;
9   }
```

Wenn wir nun (z. B. in einer Klasse namens `Schnipsel`) wie im vorigen Abschnitt die Instanzvariablen durch einfache Zugriffe der Form

```
studi.name = "Karla Karlsson";
studi.nummer = 12345;
```

setzen wollen, erhalten wir beim Übersetzen eine Fehlermeldung der Form

Konsole

```
Variable name in class Student not accessible
      from class Schnipsel.
```

Das heißt: die Zugriffe wurden verweigert.

8.2.2 Was sind Instanzmethoden?

Wie können wir aber nun Daten aus einer Klasse auslesen oder sie setzen, wenn wir hierzu überhaupt nicht berechtigt sind?
Die Antwort haben wir im vorigen Kapitel bereits angedeutet: Wir fügen der Klasse so genannte **Instanzmethoden** hinzu. Diese Methoden werden ähnlich wie in Kapitel 6 definiert:

Syntaxregel

```
public ≪RUECKGABETYP≫ ≪METHODENNAME≫ ( ≪PARAMETERLISTE≫ )
{
   // hier den auszufuehrenden Code einfuegen
}
```

Wenn Sie dies mit der Syntaxregelbox auf Seite 151 vergleichen, stellen Sie als einzigen Unterschied das Wörtchen **`static`** fest, das unserer Methodendefinition nun fehlt. Durch Weglassen dieses Wortes wird eine Methode an ein spezielles Objekt gebunden, das heißt, sie existiert nur in Zusammenhang mit einer speziellen *Instanz*. Da die Methode aber nun zu einem bestimmten Objekt gehört, hat sie auch Zugriff auf dessen spezielle Eigenschaften – also seine Instanzvariablen.
Abbildung 8.3 zeigt eine entsprechende Erweiterung unseres Klassenmodells im UML-Diagramm. Wir tragen in das untere, bislang leer gebliebene Kästchen unsere Methoden ein. Hierbei verwenden wir als Schreibweise

```
+ ≪METHODENNAME≫ ( ≪PARAMETERLISTE≫ ) : ≪RUECKGABETYP≫
```

wobei das Pluszeichen wie bei den Instanzvariablen für öffentlichen Zugriff (**`public`**) steht. Wir definieren also folgende vier Methoden:

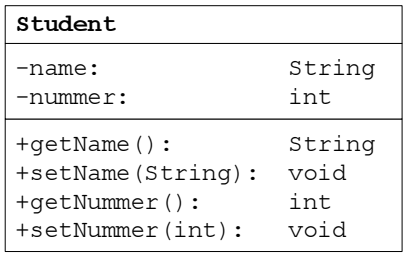

Abbildung 8.3: Die Klasse `Student`, dritte Version

- Die Methode

```
public String getName()
```

soll den Inhalt der Instanzvariablen `name` auslesen und als Resultat der Methode zurückliefern. Unser ausformulierter Java-Code lautet wie folgt:

```
/** Gib den Namen des Studenten als String zurueck */
public String getName() {
  return this.name;
}
```

Achten Sie darauf, dass wir die Instanzvariable durch **`this.`**`name` angesprochen haben. Das Schlüsselwort **`this`** liefert innerhalb eines Objektes immer eine Referenz auf das Objekt selbst. Jedes Objekt hat somit quasi eine Komponentenvariable **`this`**, die eine Referenz auf das Objekt selbst enthält. Wir können also sämtliche Instanzvariablen in der aus Abschnitt 5.2.3 bekannten Form

Syntaxregel

≪OBJEKTNAME≫.≪VARIABLENNAME≫

erreichen, indem wir für den Platzhalter ≪OBJEKTNAME≫ schlicht und ergreifend **`this`** einsetzen. Abbildung 8.4 verdeutlicht nochmals die Bedeutung der **`this`**-Referenz.

- Die Methode

```
public void setName(String name)
```

soll nun den Inhalt der Instanzvariablen `name` durch das übergebene `String`-Argument ersetzen:

```
/** Setze den Namen des Studenten auf einen bestimmten Wert */
public void setName(String name) {
  this.name = name;
}
```

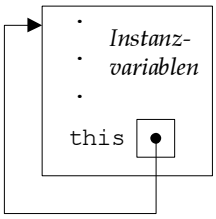

Abbildung 8.4: Die `this`-Referenz

Obwohl der Parameter `name` und die Instanzvariable `name` den gleichen Bezeichner haben, gibt es an dieser Stelle keinerlei Konflikte. Der Compiler kann beide Variablen voneinander unterscheiden, da wir die Instanzvariable mit Hilfe der **`this`**-Referenz ansprechen.

- Die Methode

```
public int getNummer()
```

liest nun den Inhalt unserer `nummer` aus und gibt ihn, genau wie bei der Methode `getName`, als Ergebnis zurück.[4] Ausformuliert lautet das wie folgt:

```
/** Gib die Matrikelnummer des Studenten als Integer zurueck */
public int getNummer() {
  return nummer;
}
```

An dieser Stelle ist zu erwähnen, dass wir in der Methode bewusst auf das Schlüsselwort **`this`** verzichtet haben. Dennoch lässt sich das Programm übersetzen. Der Grund dafür liegt darin, dass der Übersetzer in einem gewissen Ausmaß „mitdenkt". Findet er in der Methode oder den übergebenen Parametern keine Variable, die den Namen `nummer` besitzt, sucht er diese unter den Instanzvariablen.

- Zuletzt formulieren wir eine Methode

```
public void setNummer(int n)
```

zum Setzen der Instanzvariablen. Auch hier wollen wir auf die Verwendung der **`this`**-Referenz verzichten. Um mögliche Namenskonflikte zu vermeiden, haben wir dem übergebenen Parameter einen anderen Namen (`n` statt `nummer`) gegeben:

```
/** Setze die Matrikelnummer des Studenten auf einen
    bestimmten Wert */
public void setNummer(int n) {
```

[4] Hierbei mag unsere deutsch-englische Namensgebung etwas belustigend klingen, aber wir wollen von Anfang an den bestehenden Konventionen folgen, wonach Methoden, die dem Auslesen von Werten dienen, als **get-Methoden** und Methoden, die Werte einer Instanzvariablen setzen, als **set-Methoden** bezeichnet werden.

```
        nummer = n;
    }
```

Wir haben unsere Klasse `Student` nun bezüglich des Prinzips der Datenkapselung überarbeitet, indem wir sämtliche Instanzvariablen vor der Außenwelt versteckt (data hiding) und den Zugriff von außen nur noch durch get- und set-Methoden ermöglicht haben.
Am Ende dieses Abschnitts könnte man leicht vermuten, dass Instanzmethoden nicht viel mehr als einfachste Schreib/Lesemethoden sind. Wozu also das Prinzip der Datenkapselung? Steckt denn wirklich nicht mehr dahinter?
Wie so oft steckt der Teufel natürlich auch hier wieder einmal im Detail. Instanzmethoden können viel mehr als nur Werte schreiben und lesen. Wir könnten sämtliche bisher definierten Unterprogramme (vgl. Kapitel 6) als Instanzmethoden definieren, wenn wir das Wort **`static`** weglassen und sie somit an ein Objekt binden[5] – doch das verschafft uns natürlich keinen Vorteil. Die beiden folgenden Abschnitte zeigen jedoch spezielle Anwendungen, die uns die wahre Macht von Instanzmethoden demonstrieren.

8.2.3 Instanzmethoden zur Validierung von Eingaben

Die Matrikelnummer eines Studierenden ist eine von der Universitätsverwaltung vergebene Nummer, die einen Studierenden mit seiner „Akte“ identifiziert. Jeder Student bzw. jede Studentin erhält hierbei eindeutig eine solche Nummer zugeordnet. Umgekehrt ist jedoch nicht jede Zahl auch eine gültige Matrikelnummer. Um zu verhindern, dass sich Schreibfehler einschleichen oder ein Student (etwa bei Prüfungsanmeldungen) eine falsche Matrikelnummer angibt, müssen die Nummern gewisse Anforderungen, etwa bezüglich der Quersumme ihrer Ziffern, erfüllen. Eine einfache Form der Prüfung wäre etwa folgende:

> *Eine Matrikelnummer ist genau dann gültig, wenn sie fünf Stellen sowie keine führenden Nullen hat und ungerade ist.*

Um also eine ganze Zahl vom Typ **`int`** auf ihre Gültigkeit zu überprüfen, müssen wir lediglich testen,

- ob die Zahl zwischen 10000 und 99999 liegt und
- ob bei Division durch 2 ein Rest verbleibt, also `n % 2 != 0` gilt.

Diese Prüfung in eine Methode zu gießen, ist eine eher leichte Übung. Wir formulieren eine Instanzmethode `validateNummer`, wobei das Wort `validate` für „Überprüfung“ steht. Unsere Methode liefert einen **`boolean`**-Wert zurück. Ist dieser Wert **`true`**, so war die Validierung erfolgreich, d. h. wir haben eine gültige Matrikelnummer. Ist der Wert jedoch **`false`**, so haben wir eine ungültige Matrikelnummer vorliegen:

[5] In diesem Fall *müssen* wir allerdings immer ein Objekt erzeugen, um die entsprechenden Methoden aufzurufen.

```
/** Pruefe die Matrikelnummer des Studenten
    auf ihre Gueltigkeit */
public boolean validateNummer() {
  return
    (nummer >= 10000 && nummer <= 99999 && nummer % 2 != 0);
}
```

Wir können nun also unserem Studenten nicht nur eine Matrikelnummer zuweisen, sondern auch anschließend überprüfen, ob diese Nummer überhaupt gültig war. Hier stellt sich natürlich die Frage, ob unsere Klasse das nicht auch *automatisch* tun kann? Können wir nicht einfach festlegen, dass wir in unserer Klasse nur gültige Matrikelnummern hinterlegen dürfen?
Die Antwort auf diese Frage lautet wieder einmal: *Ja, das lässt sich machen!* Wir werden unsere set-Methode einfach so modifizieren, dass sie den eingegebenen Wert automatisch überprüft:

```
/** Setze die Matrikelnummer des Studenten auf einen best. Wert */
public void setNummer(int n) {
  int alteNummer = nummer;
  nummer = n;
  if (!validateNummer()) { // neue Nummer ist nicht gueltig
    nummer = alteNummer;
  }
}
```

Unsere angepasste Methode durchläuft die Prüfung in mehreren Schritten. Zuerst setzt sie die Matrikelnummer des Studenten auf den neuen Wert, speichert aber den alten Wert in der Variable `alteNummer` ab. Anschließend ruft sie die validate-Methode `validateNummer` auf. War die Validierung erfolgreich, d. h. haben wir eine gültige Matrikelnummer, so wird die Methode beendet. Andernfalls wird die alte Nummer aus `alteNummer` ausgelesen und wieder in die Instanzvariable zurückgeschrieben.
Mit unserer neuen Zugriffsmethode haben wir eine Funktionalität erreicht, die ohne Datenkapselung nicht möglich gewesen wäre. Wir weisen unserem Studenten-Objekt nicht einfach mehr eine Matrikelnummer zu, sondern überprüfen diese automatisch auf ihre Korrektheit. Eine solche Validierung kann uns in vielerlei Hinsicht von Nutzen sein; etwa, um Eingabefehler über die Tastatur zu erkennen. Das Wichtigste bei der ganzen Sache ist allerdings, dass wir für diese Erweiterung keine Veränderung an der alten Schnittstelle vornehmen mussten. Benutzer sind weiterhin in der Lage, Matrikelnummern mit `getNummer` und `setNummer` aus- und einzulesen. Programme, die vielleicht schon für die alte Klasse geschrieben waren, sind auch weiterhin lauffähig – obwohl zum Zeitpunkt der Entwicklung mit einer älteren Version gearbeitet wurde!

8.2.4 Instanzmethoden als erweiterte Funktionalität

Neben dem reinen Setzen und Auslesen von Werten können wir Instanzmethoden auch nutzen, um unseren Klassen zusätzliche Eigenschaften und Fähigkeiten zu verleihen, die sie bislang nicht besaßen.

So wollen wir etwa in diesem Abschnitt erreichen, dass Instanzen unserer Klasse eine Beschreibung ihrer selbst ausgeben können. Eine Studentin namens „Susi Sorglos“ mit der Matrikelnummer 92653 soll sich etwa in der Form

Konsole

```
Susi Sorglos (92653)
```

auf dem Bildschirm darstellen lassen.
Um diesen Zweck zu erfüllen, schreiben wir eine Methode namens `toString`, in der wir aus den Instanzvariablen eine textuelle Beschreibung generieren:

```
/** Gib eine textuelle Beschreibung dieses Studenten aus */
public String toString() {
  return name + " (" + nummer + ')';
}
```

Diese Methode kombiniert die Variablen `name` und `nummer` und erzeugt aus ihnen einen `String`. Instantiieren wir nun in unserem Hauptprogramm ein Objekt der Klasse `Student`,

```
Student studi = new Student();
studi.setName("Karla Karlsson");
studi.setNummer(12345);
```

können wir dieses Objekt durch die einfache Zeile

```
System.out.println(studi.toString());
```

auf dem Bildschirm ausgeben. Unsere Klasse ist somit in der Lage, aus ihrem inneren Zustand selbstständig eine neue Information (hier etwa eine Textbeschreibung) zu erzeugen. Unser reiner Datencontainer hat auf diese Weise ein gewisses Maß an Selbstständigkeit erreicht!
In Abschnitt 9.4 werden wir übrigens feststellen, dass für obige Bildschirmausgabe auch die Zeile

```
System.out.println(studi);
```

ausgereicht hätte. Grund hierfür ist der Umstand, dass jedes Objekt eine Methode `toString` besitzt. Wenn wir ein Objekt mit der `println`-Methode auszugeben versuchen, ruft das druckende Objekt[6] genau diese `toString`-Methode auf. In unserer Klasse `Student` haben wir diese Methode überschrieben, das heißt, wir haben mit Hilfe des Polymorphismus eine maßgeschneiderte Ausgabe für unsere Klasse modelliert.

8.3 Statische Komponenten einer Klasse

Wir haben im letzten Abschnitt mit den Instanzvariablen und -methoden ein wichtiges Gebiet des objektorientierten Programmierens kennen gelernt. Die

[6] Auch die Methode `println` ist Instanzmethode eines Objektes, des so genannten Ausgabestroms. Das Objekt `System.out` ist ein solcher Strom.

Möglichkeit, Variablen oder sogar ganze Methoden einem bestimmten Objekt zuordnen zu können, hat uns Perspektiven erschlossen, die wir mit unseren bisherigen Programmiererfahrungen nicht sahen.
Hier stellt sich jedoch die Frage, wie sich das früher Gelernte mit diesen neuen Technologien vereinbaren lässt. Instanzmethoden ähneln vom Aufbau her zwar unseren Methoden aus Kapitel 6, sind aber schon insofern vollkommen verschieden, als sie zu einem speziellen Objekt gehören. Müssen wir also unser ganzes Wissen über Bord werfen?
Natürlich nicht! Aus objektorientierter Sicht handelt es sich bei unseren früher verwendeten Methoden um die so genannten **Klassenmethoden**, auch **statische Methoden** genannt. In diesem Kapitel haben wir bisher nur Instanzmethoden definiert – also Methoden, die einer ganz bestimmten *Instanz* einer Klasse gehören. Klassenmethoden wiederum folgen dem gleichen Schema. Statt einer einzelnen Instanz gehören sie allerdings der gesamten *Klasse*, das heißt, alle Objekte teilen sich eine einzige Methode. Diese Methode existiert vielmehr sogar, wenn *kein einziges Objekt* zu unserer Klasse existiert.
Unsere früheren Programme haben diesen Umstand ausgenutzt, um Ihnen als Anfänger die objektorientierte Sichtweise zu ersparen. Wir haben Klassen definiert (jedes unserer Programme war eine Klassendefinition) und diese nur mit Klassenmethoden gefüllt. Obwohl wir nie eine Instanz dieser Klassen erzeugt haben, konnten wir die einzelnen Methoden problemlos aufrufen. Jetzt, da Sie im Begriff sind, ein OO-Profi zu werden, wissen Sie es natürlich besser. Nehmen Sie eines Ihrer alten Programme, und versuchen Sie, mit Hilfe des **`new`**-Operators eine Instanz zu bilden. Es wird Ihnen gelingen.

8.3.1 Klassenvariablen und -methoden

Am ehesten wird der Nutzen von statischen Komponenten deutlich, wenn wir mit einem konkreten Anwendungsfall beginnen. Unsere Klasse `Student` besitzt momentan zwei Datenelemente, nämlich den Namen und die Matrikelnummer des Studenten bzw. der Studentin.
Aus statistischer Sicht mag es vielleicht interessant sein, die Zahl der instantiierten Studentenobjekte zu zählen. Wird beispielsweise eine neue Universität eröffnet und verwendet diese von Anfang an unsere Studentenverwaltung, so könnte man aus dieser Variablen erfahren, wie viele Studierende es im Laufe der Geschichte an dieser Universität gegeben hat.
Nun stehen wir jedoch vor dem Problem, dass wir diese Variable – wir wollen sie der Einfachheit halber einmal `zaehler` nennen – keiner speziellen Instanz unserer Klasse zuordnen können. Vielmehr handelt es sich hierbei um eine Eigenschaft, die zu der Gesamtheit *aller* Studentenobjekte gehört. Die Anzahl aller Studenten macht keine Aussage über einen speziellen Studenten, sondern über die Studenten an sich. Sie sollte daher *allen* Studenten angehören, sprich, eine **statische Komponente** der Klasse `Student` sein.

Wir erzeugen deshalb eine Variable, die keiner bestimmten Instanz, sondern der gesamten Klasse gehört, gemäß der folgenden Regel:[7]

Syntaxregel

```
private static ≪TYP≫ ≪VARIABLENNAME≫ = ≪INITIALWERT≫;
```

Wir stellen fest, dass sich die Definition von Klassenvariablen nicht sehr von dem unterscheidet, was wir in Abschnitt 5.2 über Instanzvariablen gelernt haben. Mit Hilfe des Wortes **`private`** schützen wir unsere Variable vor Zugriffen von außerhalb. Typ, Variablenname und Initialwert sind uns ebenfalls bekannt und würden im Fall unseres Zählers zu folgender Definition führen:

```
private static int zaehler = 0;
```

Neu ist für uns an dieser Stelle lediglich das Schlüsselwort **`static`**, das wir bislang nur aus unseren Methoden im ersten Teil des Buches kannten. Dieses Wort weist eine Variable oder Methode als statische Komponente einer Klasse aus. Wenn wir eine Variable also als **`static`** beschreiben, gehört sie allen Instanzen einer Klasse zugleich. Wir können den Inhalt der Variablen auslesen, indem wir eine entsprechende get-Methode definieren:

```
/** Gib die Zahl der erzeugten Studentenobjekte zurueck */
public static int getZaehler() {
  return zaehler;
}
```

Beachten Sie hierbei, dass wir auch bei dieser Methode das Schlüsselwort **`static`** verwendet haben, die Methode also der Klasse, nicht den Objekten zugeordnet haben. Die Methode `getZaehler` ist also eine Klassenmethode, die wir etwa durch einen Aufruf der Form

```
System.out.println(Student.getZaehler());
```

aus jedem beliebigen Programm aufrufen können, ohne eine konkrete Referenz auf ein Studentenobjekt zu besitzen.
Wie können wir aber nun ein Objekt so erzeugen, dass der interne (private) Zähler korrekt erhöht wird? Zu diesem Zweck entwerfen wir eine Methode `createStudent`, die uns ein neues Studentenobjekt erzeugt. Auch diese Methode müssen wir statisch machen, da sie schließlich gerade zum Erzeugen von Objekten benutzt werden soll, also nicht aus einem Objekt heraus aufgerufen wird:

```
/** Erzeugt ein neues Studentenobjekt */
public static Student createStudent() {
  zaehler++; // erhoehe den Zaehler
  return new Student();
}
```

Unsere Methode zählt bei Aufruf zuerst die Variable `zaehler` hoch und aktualisiert somit deren Stand. Im zweiten Schritt wird mit Hilfe des **`new`**-Operators ein

[7] Der initiale Wert könnte an dieser Stelle auch wegfallen.

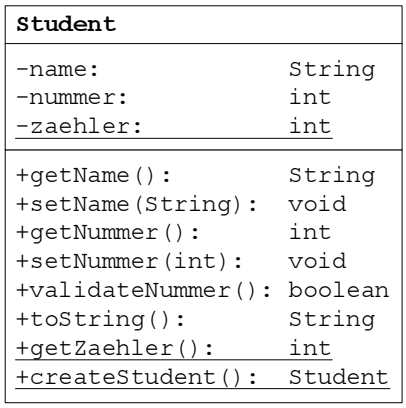

Abbildung 8.5: Die Klasse `Student`, mit Objektzähler

neues Objekt erzeugt und dieses als Ergebnis zurückgegeben. Nun können wir in unseren Programmen Studentenobjekte durch einen einfachen Methodenaufruf erzeugen lassen und somit den Zähler korrekt aktualisieren:

```
Student studi = Student.createStudent();
System.out.println(Student.getZaehler());
```

Leider hat diese Methode, neue Studentenobjekte zu erzeugen, einen gewaltigen Pferdefuß: bei älteren Programmen, die ihre Objekte noch mit Hilfe des **`new`**-Operators erzeugen, funktioniert der Zähler nicht korrekt. Wir laufen auch immer Gefahr, dass andere Programmierer, die unsere Klasse `Student` benutzen, den Fehler begehen, Objekte direkt zu erzeugen. Wir werden in Abschnitt 8.4.1 jedoch eine Methode kennen lernen, diese Probleme auf elegante Art und Weise zu lösen.
Jetzt werfen wir noch einen Blick auf unsere gewachsene Klasse `Student` im UML-Klassendiagramm (Abbildung 8.5). Klassenmethoden und Klassenvariablen werden im UML-Diagramm durch Unterstreichung gekennzeichnet. Wir stellen fest, dass wir – obwohl unsere Klasse inzwischen beträchtlich gewachsen ist – durch die Grafik noch immer einen schnellen Überblick über die Komponenten erhalten, aus denen sich die Klasse zusammensetzt. Oft ist es sinnvoll, private Variablen nicht in das UML-Diagramm einzuzeichnen, denn für den Entwurf eines Systems von Klassen (hierzu dient uns UML) ist es letztendlich ausreichend zu wissen, welche Schnittstelle eine Klasse nach außen zu bieten hat. Dadurch lassen sich große Klassen übersichtlicher gestalten. Auch wir wollen nachfolgend gelegentlich von dieser Regel Gebrauch machen.

8.3.2 Klassenkonstanten

Wie wir aus Abschnitt 4.4.1 wissen, ist es möglich, mit Hilfe des Schlüsselwortes **`final`** aus „normalen“ Variablen **`final`**-Variablen zu machen, sie also zu sym-

bolischen Konstanten werden zu lassen. Das gilt natürlich nicht nur für lokale Variablen innerhalb einer Methode, sondern auch für Klassenvariablen, die durch das vorangestellte **`final`** zu Klassenkonstanten werden.
Konstanten werden in Java häufig dann eingesetzt, wenn man eine nichtssagende Codierung durch eine selbst erklärende Begrifflichkeit erklären will oder wenn man schwer zu merkende Werte wie etwa den Wert der mathematischen Konstanten π (gesprochen „pi“, etwa $3.14\ldots$) benennen will. Hierbei gilt ja als Konvention, dass wir Konstanten in unseren Programmen immer groß schreiben. Im Falle von π verwendet Java die Bezeichnung `PI`. Da diese Konstante in der Klasse `Math` deklariert ist, können wir sie bekanntlich über `Math.PI` ansprechen.

8.4 Instantiierung und Initialisierung

In diesem Abschnitt beschäftigen wir uns mit der Frage, wie wir Einfluss auf den Erzeugungsprozess eines Objektes nehmen können. Bereits auf Seite 214 hatten wir festgestellt, dass es uns gelingen müsste, in irgendeiner Form Einfluss auf den **`new`**-Operator zu nehmen. Unsere Methode `createStudent` und der besagte Operator taten schließlich nicht mehr das Gleiche; nur die `create`-Methode zählte unseren Zähler korrekt hoch.
Nun lernen wir Mittel und Wege kennen, unser Vorhaben in die Tat umzusetzen.

8.4.1 Konstruktoren

Erinnern wir uns: Bevor wir die Methode `createStudent` erschufen, hatten wir unsere Objekte durch eine Zeile der Form

```
Student studi = new Student();
```

instantiiert, wobei der so genannte **`new`**-Operator (wie bereits auf Seite 135 beschrieben) nach der Regel

Syntaxregel

```
≪INSTANZNAME≫ = new ≪KLASSENNAME≫ ();
```

angewendet wurde.
Wenn wir uns diese Zeile etwas genauer ansehen, so fallen uns die runden Klammern am Ende auf. Diese Klammern kennen wir bislang nur vom Aufruf von Methoden her! Ruft die Verwendung des **`new`**-Operators etwa ebenfalls eine Methode auf?
Tatsächlich ist der Vorgang des „Erbauens“ eines Objektes etwas komplizierter. In Abschnitt 8.4.4 gehen wir auf die tatsächlichen Mechanismen näher ein. Wir können aber an dieser Stelle schon vereinfacht sagen, dass am Ende dieses Vorganges tatsächlich eine Art von Methode aufgerufen wird: der so genannte **Konstruktor**.

Konstruktoren sind keine Methoden im eigentlichen Sinn, da sie nicht – wie etwa Klassen- oder Instanzmethoden – explizit aufgerufen werden. Sie haben auch keinen Rückgabetyp (nicht einmal **void**). Die Definition des Konstruktors erfolgt nach dem Schema:[8]

Syntaxregel

```
public ≪KLASSENNAME≫ ( ≪PARAMETERLISTE≫ )
{
   // hier den auszufuehrenden Code einfuegen
}
```

Aus dieser Regel schließen wir zwei wichtige Dinge:

1. Der Konstruktor heißt immer so wie die Klasse.
2. Der Konstruktor verfügt über eine Parameterliste, in der wir Argumente vereinbaren können (was wir im nächsten Abschnitt auch tun werden).

Mit dieser einfachen Regel können wir nun also Einfluss auf die Erzeugung unseres Objektes nehmen – genau das wollen wir auch tun. Wir beginnen mit dem einfachsten Fall: einem Konstruktor, der keinerlei Argumente besitzt und absolut nichts tut:

```
public Student() {}
```

Dieser Konstruktor, manchmal auch als **Standard-Konstruktor** oder **Default-Konstruktor** bezeichnet, wurde bisher vom Übersetzer automatisch erzeugt. Er wird vom System aufgerufen, wenn wir z. B. mit

```
Student studi = new Student();
```

ein Objekt instantiieren. Der Standardkonstruktor wird nur angelegt, wenn man keine eigenen Konstruktoren anlegt – und nur dann! Wenn wir also im Folgenden eigene Konstruktoren für unsere Klassen definieren, wird für diese vom System kein Standardkonstruktor mehr angelegt.
Der folgende Konstruktor aktualisiert unsere Klassenvariable `zaehler`, indem er sie automatisch um den Wert 1 erhöht:

```
/** Argumentloser Konstruktor */
public Student() {
  zaehler++;
}
```

Wenn wir nun mit Hilfe des **new**-Operators ein Studentenobjekt erzeugen, so wird durch den Aufruf des Konstruktors der Zähler automatisch aktualisiert. Wir können uns also die zusätzliche Erhöhung in unserer `createStudent`-Methode sparen:

[8] Hierbei kann man statt **public** natürlich auch andere Zugriffsrechte vergeben.

```
/** Erzeugt ein neues Studentenobjekt */
public static Student createStudent() {
  return new Student();
}
```

Tatsächlich stellen wir fest, dass es nun wieder keinen Unterschied mehr bedeutet, ob wir unsere Objekte mit **new** oder mit `createStudent` erzeugen. Der Prozess der Instantiierung wurde somit vereinheitlicht, die auf Seite 214 angemahnte Abwärtskompatibilität[9] wiederhergestellt.

8.4.2 Überladen von Konstruktoren

Wir wollen neben den bisher vorhandenen Daten eine weitere Instanzvariable definieren: In der ganzzahligen Variable `geburtsjahr` möchten wir das Jahr hinterlegen, in dem der betreffende Student bzw. die betreffende Studentin geboren wurde.

```
/** Geburtsjahr eines Studenten */
private int geburtsjahr;
```

Die Variable `geburtsjahr` soll im Gegensatz zu unseren bisherigen Instanzvariablen jedoch eine Besonderheit besitzen. Wir definieren zwar eine get-Methode, mit der wir den Wert der Variablen auslesen können

```
/** Gib das Geburtsjahr des Studenten als Integer zurueck */
public int getGeburtsjahr() {
  return geburtsjahr;
}
```

formulieren aber keine set-Methode, um den entsprechenden Wert zu setzen bzw. zu verändern. Der Grund hierfür ist relativ einfach. Alle bisher definierten Werte können sich ändern. Der Student bzw. die Studentin kann heiraten und den Namen seines Partners annehmen. Er kann sein Studienfach oder die Universität wechseln, was den Inhalt der Variablen `fach` und `nummer` beeinflussen würde. Nur eines kann unser(e) Student(in) niemals verändern: das Jahr, in dem er bzw. sie geboren wurde.
Wir wollen also den Inhalt der Variablen beim Erzeugen festlegen. Danach soll diese Variable von außen nicht mehr verändert werden können. Im Fall unseres argumentlosen Konstruktors sähe dies etwa wie folgt aus:

```
/** Argumentloser Konstruktor */
public Student() {
  zaehler++;
  geburtsjahr = 1970;
}
```

Wir setzten also den Inhalt unserer Variablen auf einen Standardwert, das Jahr 1970, was natürlich insbesondere deshalb unbefriedigend ist, weil nur ein geringer Teil der heute Studierenden in diesem Jahr geboren wurde. Deshalb definieren

[9] Dies bedeutet, dass Programme, die für ältere Versionen unserer Klasse `Student` geschrieben wurden, auch mit unserer neuen Version funktionieren.

wir einen zweiten Konstruktor, in dem wir das Geburtsjahr als einen Parameter übergeben:

```
/** Konstruktor, bei dem sich das Geburtsjahr setzen laesst. */
public Student(int geburtsjahr) {
  zaehler++;
  this.geburtsjahr = geburtsjahr;
}
```

Wir haben unseren Konstruktor also **überladen**, wie wir es schon in Abschnitt 6.1.5 mit Methoden gemacht haben. Analog dazu unterscheidet Java auch die Konstruktoren einer Klasse

- anhand der *Zahl* der Argumente,
- anhand des *Typs* der Argumente und
- anhand der *Position* der Argumente.

Wir können beim Überladen also den gleichen Regeln folgen – unsere Definition des zweiten Konstruktors war somit korrekt – und ihn wie gewohnt verwenden, indem wir das Geburtsjahr innerhalb der Klammern des **new**-Operators mit aufführen. So generiert etwa die folgende Zeile einen im Jahr 1982 geborenen Studenten:

```
Student studi = new Student(1982);
```

In den Übungsaufgaben beschäftigen wir uns noch einmal mit dem Überladen von Konstruktoren. Da Sie diesen Mechanismus jedoch bereits von den Methoden her kennen, stellt er bei Weitem kein Hexenwerk mehr dar.
An diesem Punkt jedoch noch eine kleine Anmerkung, die die Programmierung insbesondere von vielen Konstruktoren in einer Klasse vereinfacht. Wenn wir einen Blick auf unsere beiden Konstruktoren werfen, so stellen wir fest, dass sich diese in ihrer Struktur sehr ähneln:

```
/** Argumentloser Konstruktor */
public Student() {
  zaehler++;
  geburtsjahr = 1970;
}

/** Konstruktor, bei dem sich das Geburtsjahr setzen laesst. */
public Student(int geburtsjahr) {
  zaehler++;
  this.geburtsjahr = geburtsjahr;
}
```

Beide Konstruktoren erhöhen zuerst den Zähler und setzen dann die Variable `geburtsjahr` auf einen vorbestimmten Wert. Unser argumentloser Konstruktor ist hierbei gewissermaßen ein „Spezialfall“ des anderen Konstruktors, da er das Geburtsjahr nicht übergeben bekommt, sondern auf einen festen Wert setzt. Wir können diesen Konstruktor also einfacher formulieren, indem wir ihn auf seinen „großen Bruder“ zurückführen:

```
public Student() {
  this(1970);
}
```

Hierbei verwenden wir das Schlüsselwort **this**, um einen Konstruktor aus einem anderen Konstruktor heraus aufzurufen. Dieser Vorgang kann nur innerhalb von Konstruktoren und auch dort nur einmal geschehen – nämlich *als allererster Befehl innerhalb des Konstruktors*. Dieser eine erlaubte Aufruf gestattet es uns jedoch, nicht jede einzelne Codezeile doppelt formulieren zu müssen. Insbesondere bei großen und aufwendigen Konstruktoren erspart uns das eine Menge Arbeit.

8.4.3 Der statische Initialisierer

Spätestens seit Gaston Leroux' Erfolgsroman wissen wir es alle: eine wirklich erfolgreiche Institution benötigt ein *Phantom*. Angefangen mit dem Phantom der (Pariser) Oper übertrug sich dieser Trend mittels Hollywoodstreifen auf Filmstudios, Krankenhäuser und sonstige öffentliche Gebäude.
Wir wollen dieser Entwicklung Rechnung tragen und auch unserer Universität ein Phantom spendieren. Dieses Phantom soll eine konstante Klassenvariable sein und unter dem Namen `Student.PHANTOM` angesprochen werden können:

```
/** Diese Konstante repraesentiert
    das Phantom des Campus */
public static final Student PHANTOM;
```

Unser Phantom soll die Matrikelnummer -12345 besitzen, auf den Namen „Erik le Phant" hören und im Jahr 1735 geboren sein. Ferner soll er offiziell gar nicht existieren, das heißt, seine Existenz soll den Studentenzähler nicht beeinflussen.
An dieser Stelle bekommen wir mit der Initialisierung unserer Konstanten anscheinend massive Probleme:

1. Die Konstante `Student.PHANTOM` soll zusammen mit der Klasse existieren, ohne dass wir sie in unserem Hauptprogramm erst in irgendeiner Form initialisieren müssen.

2. Die Zahl -12345 ist keine gültige Matrikelnummer. Unsere `setNummer`-Methode würde diesen Wert nicht als gültige Eingabe akzeptieren. Wir können diesen Wert also von außen nicht setzen.

3. Jedes Mal, wenn wir mit dem **new**-Operator ein Objekt erzeugen, wird die interne Variable `zaehler` automatisch hochgezählt. Da wir aber von außen nur lesenden Zugriff auf den Zähler haben, können wir diesen Umstand nicht rückgängig machen.

Wie wir sehen, kommen wir an dieser Stelle mit einer Initialisierung „von außen" nicht weiter. Wir benötigen eine Möglichkeit, statische Komponenten einer Klasse beim Systemstart[10] automatisch zu initialisieren. Hierfür verwenden wir den so

[10] Genauer gesagt, wenn wir die Klasse zum ersten Mal verwenden.

genannten **statischen Initialisierer**, umgangssprachlich oft einfach **`static`**-Block genannt.[11]
Statische Initialisierer werden nach folgender Regel erschaffen:

Syntaxregel

```
static
{
   // hier den auszufuehrenden Code einfuegen
}
```

In einer Klasse können beliebig viele static-Blöcke auftreten. Sobald die Klasse dem Java-System bekannt gemacht wird (das so genannte Laden der Klasse), werden die static-Blöcke in der Reihenfolge ausgeführt, in der sie im Programmcode auftauchen. Hierbei gelten die folgenden wichtigen Regeln:

- *Statische Initialisierer haben nur Zugriff auf statische Komponenten einer Klasse.* Sie können keine Instanzvariablen manipulieren, da diese nur innerhalb von Objekten existieren. Natürlich mit der Ausnahme, dass Sie innerhalb des static-Blocks ein Objekt, mit dem Sie arbeiten wollen, erzeugt haben.
- *Statische Initialisierer haben Zugriff auf alle (auch private) Teile einer Klasse.* Im Gegensatz zu einer Initialisierung „von außen" befinden wir uns beim static-Block innerhalb der Klasse. Wir können selbst die für andere unsichtbaren Bereiche einsehen und manipulieren.
- *Statische Initialisierer haben nur Zugriff auf statische Komponenten, die im Programmcode* vor *ihnen definiert wurden.* Wenn Sie also eine statische Variable durch einen static-Block initialisieren wollen, muss der static-Block *nach* der Definition der Klassenvariable erfolgen.

Wir wollen diese Regeln nun berücksichtigen und unsere Konstante initialisieren. Hierzu erzeugen wir einen static-Block, den wir (um bezüglich der Reihenfolge auf Nummer sicher zu gehen) an das Ende unserer Klassendefinition setzen:

```
/* =========================
   STATISCHE INITIALISIERUNG
   =========================
 */

static {
  // Erzeuge das PHANTOM-Objekt
  PHANTOM = new Student(1735);
  PHANTOM.name = "Erik le Phant";
  PHANTOM.nummer = -12345;
  // Setze den Zaehler wieder zurueck
  zaehler = 0;
}
```

[11] Die offizielle englischsprachige Bezeichnung aus der Java Language Specification ist übrigens **static initializer**.

Gehen wir nun die einzelnen Zeilen unseres statischen Initialisierers genauer durch. In der ersten Zeile

```
PHANTOM = new Student(1735);
```

haben wir mit Hilfe des **new**-Operators ein neues Studentenobjekt (mit Geburtsdatum 1735) erzeugt und der Konstanten `PHANTOM` zugewiesen. Unsere Konstante ist somit belegt und kann nicht mehr verändert werden.
In der folgenden Zeile werden wir nun anscheinend gegen diesen Grundsatz verstoßen. Wir nutzen unseren direkten Zugriff auf die private Instanzvariable `name` aus und setzen ihren Inhalt auf den Namen „Erik le Phant“:

```
PHANTOM.name = "Erik le Phant";
```

Haben wir somit gegen das Gesetz, finale Variablen nicht mehr verändern zu können, verstoßen? Die Antwort lautet *nein*, und ihre Begründung liegt wieder einmal in dem Umstand, dass es sich bei Klassen um Referenzdatentypen handelt. In unserer finalen Variablen `PHANTOM` steht nämlich nicht das Objekt selbst, sondern eine *Referenz*, also ein Verweis auf das tatsächliche Objekt. Diese Referenz ist konstant, das heißt, unsere Variable wird immer auf ein und dasselbe Studentenobjekt verweisen. Das Objekt selbst ist jedoch ein ganz „normaler“ Student und kann als solcher von uns auch manipuliert[12] werden.
In der folgenden Zeile nutzen wir unseren Zugriff auf private Komponenten aus, um den Wert der Matrikelnummer auf -12345 zu setzen:

```
PHANTOM.nummer = -12345;
```

Da wir hierbei den Wert der Variablen direkt setzen, also nicht über die set-Methode gehen, wird die `validate`-Methode für unsere Variable `nummer` nicht aufgerufen. Wir können den Inhalt unserer Variablen somit ungestört auf einen (eigentlich nicht erlaubten) Wert setzen.
Nun kümmern wir uns noch um den statischen Objektzähler. Dass der **new**-Operator unsere Variable `zaehler` auf den Wert 1 gesetzt hat, konnten wir nicht verhindern. Wir machen dies im Nachhinein jedoch wieder rückgängig, indem wir unseren Objektzähler einfach wieder auf null setzen:

```
zaehler = 0;
```

Wir haben innerhalb weniger Zeilen einen statischen Initialisierer geschaffen, der

1. die Konstante `Student.PHANTOM` automatisch initialisiert, sobald die Klasse benutzt wird,
2. die Matrikel`nummer` auf den (eigentlich inkorrekten) Wert -12345 setzt und somit die automatische Prüfung umgeht und
3. den `zaehler` wieder zurücksetzt, sodass unser Phantom in der Objektzählung nicht erscheint.

Unsere Probleme sind also gelöst.

[12] Natürlich lehnen wir jegliche Manipulation von Studierenden grundsätzlich ab. Das Beispiel dient lediglich zu Ausbildungszwecken und erfolgt auch nur an unserem Phantom.

8.4.4 Der Mechanismus der Objekterzeugung

Wir haben in den letzten Abschnitten verschiedene Mechanismen kennen gelernt, um Klassen- und Instanzvariablen mit Werten zu belegen. Unsere Konstruktoren spielen hierbei eine wichtige Rolle, sind aber nicht die einzigen wichtigen Bestandteile des Instantiierungsprozesses. Wenn wir beispielsweise unserer Variablen `name` in ihrer Definition

```
private String name = "DummyStudent";
```

einen Initialisierer hinzufügen und ferner im Konstruktor die Zeile

```
this.name = "Namenlos";
```

hinzufügen – auf welchen Wert wird unser Studentenname bei der Initialisierung dann gesetzt? Ist er dann „Namenlos“ oder ein „DummyStudent“?
Um diese Frage beantworten zu können, sollte man (zumindest in groben Zügen) den Mechanismus verstehen, mit dem unsere Objekte erzeugt werden. Wir werden uns deshalb in diesem Abschnitt näher damit beschäftigen. Zu diesem Zweck betrachten wir zwei einfache Klassen, die in Abbildung 8.6 skizziert sind.

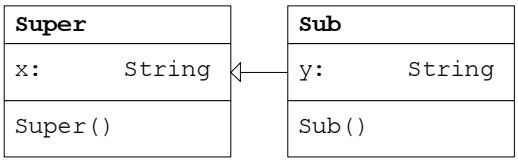

Abbildung 8.6: Beispielklassen für Abschnitt 8.4.4

Die Klassen `Super` und `Sub` stehen in einer verwandtschaftlichen Beziehung zueinander: `Sub` ist die Subklasse von `Super`. Sie erbt somit deren Eigenschaften, das heißt, in diesem Fall die öffentliche Instanzvariable `x`. Ferner wird in `Sub` eine zweite Instanzvariable namens `y` definiert, die also die Funktionalität der Superklasse um ein weiteres Datum ergänzt. Im Folgenden werden wir uns mit der Frage beschäftigen, welche Aktionen innerhalb des Systems beim Aufruf eines Konstruktors[13] der Subklasse in der Form

```
new Sub();
```

ausgelöst werden.
Wir betrachten erst einmal die Theorie. Ein Objekt wird vom System in den folgenden Schritten angelegt:

1. Das System organisiert Speicherplatz, um den Inhalt sämtlicher Instanzvariablen abspeichern zu können, die innerhalb des Objektes benötigt werden. In unserem Fall wären das für ein `Sub`-Objekt also die Variablen `x`

[13] Die Konstruktoren werden im UML-Diagramm wie Methoden dargestellt, allerdings lässt man den Rückgabetyp weg. Jede unserer beiden Klassen besitzt also einen argumentlosen Konstruktor.

und `y`. Sollte nicht genug Speicher vorhanden sein, entsteht ein so genannter `OutOfMemory`-Fehler, der das gesamte Java-System zum Absturz bringen kann. In Ihren Programmen wird dies aber normalerweise nicht der Fall sein.

2. Die Instanzvariablen werden mit ihren Standardwerten (Default-Werten, gemäß Tabelle 8.1) belegt.

Datentyp	Standardwert
byte	(**byte**)0
short	(**short**)0
int	0
long	0L
float	0.0f
double	0.0d
char	(**char**)0
boolean	**false**
Referenzdatentyp	**null**

Tabelle 8.1: Default-Werte von Instanzvariablen

3. Der Konstruktor wird mit den übergebenen Werten aufgerufen. Hierbei wird in Java nach dem folgenden System vorgegangen:

   (a) Ist die erste Anweisung des Konstruktorrumpfes *kein* Aufruf eines anderen Konstruktors (also weder **this**`(...)` noch **super**`(...)`), so wird implizit der Aufruf des Standard-Konstruktors der direkten Superklasse **super**`()` ergänzt und auch aufgerufen. Unmittelbar nach diesem impliziten Aufruf werden alle in der Klasse mit Initialisierern deklarierten Instanzvariablen mit den entsprechenden Werten initialisiert. Haben wir etwa in unserer Klasse `Sub` die Variable `y` in der Form

   ```
   public String y = "vor Sub-Konstruktor";
   ```

   definiert, lautet der Wert von `y` nun also `vor Sub-Konstruktor`. Erst danach werden die restlichen Anweisungen des Konstruktorrumpfes ausgeführt. Auf das Schlüsselwort **super** gehen wir im nächsten Kapitel noch genauer ein.

   (b) Ist die erste Anweisung innerhalb des Konstruktorrumpfes von der Form **super**`(...)`, wird der entsprechende Konstruktor der direkten Superklasse aufgerufen. Danach werden alle in der Klasse mit Initialisierern deklarierten Instanzvariablen mit den entsprechenden Werten initialisiert und die restlichen Anweisungen des Konstruktorrumpfes ausgeführt.

   (c) Ist die erste Anweisung innerhalb des Konstruktorrumpfes von der Form **this**`(...)`, wird der entsprechende Konstruktor derselben Klasse aufgerufen. Danach sind alle in der Klasse mit Initialisierern deklarierten Instanzvariablen bereits initialisiert, und es werden nur noch die restlichen Anweisungen des Konstruktorrumpfes ausgeführt.

Wir werden diese Regeln nun an unserem konkreten Beispiel anzuwenden versuchen. Hierfür werfen wir zunächst einen Blick auf die Definition unserer beiden Klassen in Java:

```
1  public class Super {
2
3    /** Eine oeffentliche Instanzvariable */
4    public String x = "vor Super-Konstruktor";
5
6    /** Ein argumentloser Konstruktor */
7    public Super() {
8      System.out.println("Super-Konstruktor gestartet.");
9      System.out.println("x = " + x);
10     x = "nach Super-Konstruktor";
11     System.out.println("Super-Konstruktor beendet.");
12     System.out.println("x = " + x);
13   }
14 }
```

Unsere Klasse `Sub` leitet sich hierbei von der Klasse `Super` ab, was wir in Java durch das Schlüsselwort **`extends`** zum Ausdruck bringen. Der restliche Aufbau der Klasse ergibt sich auch aus dem dazugehörigen UML-Diagramm 8.6:

```
1  public class Sub extends Super {
2
3    /** Eine weitere oeffentliche Instanzvariable */
4    public String y = "vor Sub-Konstruktor";
5
6    /** Ein argumentloser Konstruktor */
7    public Sub() {
8      System.out.println("Sub-Konstruktor gestartet.");
9      System.out.println("x = " + x);
10     System.out.println("y = " + y);
11     x = "nach Sub-Konstruktor";
12     y = "nach Sub-Konstruktor";
13     System.out.println("Sub-Konstruktor beendet.");
14     System.out.println("x = " + x);
15     System.out.println("y = " + y);
16   }
17 }
```

Wenn wir nach dem allgemeinen Muster vorgehen, unterteilt sich der Instantiierungsvorgang in verschiedene Schritte. Wir haben den Ablauf in neun Einzelschritte zerlegt, die in Abbildung 8.7 grafisch dargestellt sind:

1. Im Speicher wird Platz für ein Objekt der Klasse `Sub` reserviert. Es werden die Instanzvariablen `x` und `y` angelegt und mit den Default-Werten initialisiert.
2. Der Konstruktor wird aufgerufen. Da wir in unserem Code nicht explizit mit **`super`** gearbeitet haben, ruft das System automatisch den argumentlosen Konstruktor der Superklasse auf. Bei dessen Ablauf wird zunächst (automatisch) die Variable `x` initialisiert.

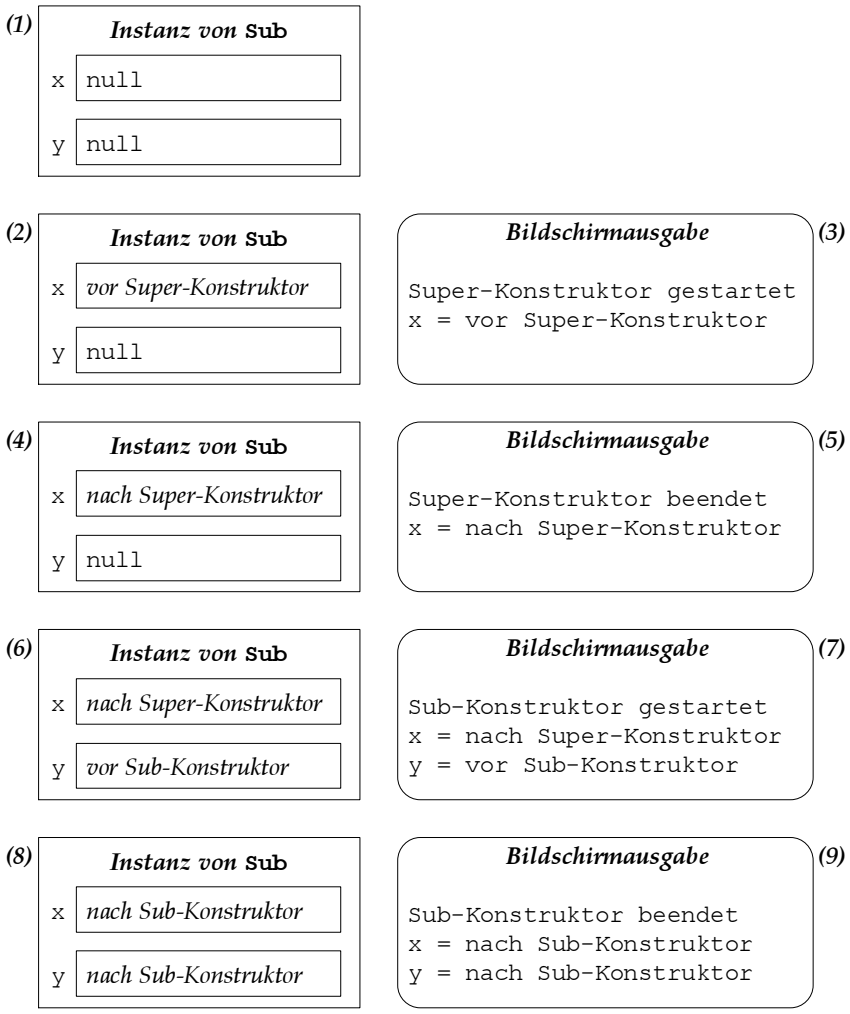

Abbildung 8.7: Instantiierungsprozess von Sub- und Superklasse

3. Im weiteren Ablauf des Super-Konstruktors wird eine Meldung auf dem Bildschirm ausgegeben (durch Zeile 8 und 9 im Programmcode).
4. Danach wird der Inhalt der Variable `x` auf den Wert „nach Super-Konstruktor“ gesetzt.
5. Bevor der Konstruktor der Superklasse beendet wird, gibt er eine entsprechende Meldung auf dem Bildschirm aus (Zeile 11 bis 13). Der Konstruktor der Super-Klasse wurde ordnungsgemäß beendet.

6. Nun wird der Konstruktor der Klasse `Sub` fortgesetzt mit der (automatischen) Initialisierung von `y`, d. h. die Variable wird auf „vor Sub-Konstruktor“ gesetzt.
7. Nun erfolgt die eigentliche Ausführung unseres Konstruktors der Klasse `Sub`. Zu Beginn des Konstruktors wird eine entsprechende Meldung ausgegeben; die Variablen `x` und `y` haben die Werte „nach Super-Konstruktor“ bzw. „vor Sub-Konstruktor“.
8. Zuletzt werden die Variablen `x` und `y` wiederum auf einen neuen Wert gesetzt (Zeile 11 und 12 im Programmtext der Klasse `Sub`).
9. In der anschließenden Bildschirmausgabe wird uns diese Veränderung bestätigt.

Die komplette Ausgabe unseres Programms lautet also wie folgt:

Konsole

```
Super-Konstruktor gestartet.
x = vor Super-Konstruktor
Super-Konstruktor beendet.
x = nach Super-Konstruktor

Sub-Konstruktor gestartet.
x = nach Super-Konstruktor
y = vor Sub-Konstruktor
Sub-Konstruktor beendet.
x = nach Sub-Konstruktor
y = nach Sub-Konstruktor
```

Wie wir sehen, haben unsere Variablen während des Instantiierungsprozesses bis zu drei verschiedene Werte angenommen. Wir können diese Zahl beliebig steigern, indem wir die Zahl der sich voneinander ableitenden Klassen erhöhen. In jeder Superklasse können wir einen Konstruktor definieren, der den Wert einer Instanzvariable verändert.
Im Allgemeinen ist es natürlich nicht sinnvoll, seine Programme auf diese Weise zu verfassen – der Quelltext wird dann unleserlich und ist schwer nachzuvollziehen. Das Wissen um den Instantiierungsprozess hilft uns jedoch weiter, um etwa die Eingangsfrage unseres Abschnitts bezüglich der Klasse `Student` beantworten zu können. Machen Sie sich anhand der Regeln klar, warum die richtige Antwort „Namenlos“ lautet.

8.5 Zusammenfassung

Wir haben anhand eines einfachen Anwendungsfalles – der Klasse `Student` – die grundlegenden Mechanismen kennen gelernt, um in Java mit Klassen umzugehen. Wir haben Instanzvariablen und Instanzmethoden kennen gelernt – Variablen und Methoden also, die direkt einem Objekt zugeordnet sind. Dieses neue

Konzept stand im Gegensatz zu unserer bisherigen Vorgehensweise, Methoden als statische Komponenten einer Klasse zu erklären. Die Verwendung dieser statischen Komponenten, also Klassenvariablen und Klassenmethoden, haben wir dennoch nicht vollständig verworfen, sondern anhand eines einfachen Beispiels (der Variablen `zaehler`) ihren praktischen Nutzen in der Objektorientierung demonstriert.
Wir haben die Schlüsselworte **`public`** und **`private`** kennen gelernt, mit deren Hilfe wir Teile einer Klasse öffentlich machen oder vor der Außenwelt verstecken konnten. Dabei haben wir gelernt, wie man dem Prinzip der Datenkapselung entspricht, indem wir Variablen privat deklariert und Lese- und Schreibzugriff über entsprechende (öffentliche) Methoden gewährt haben. Auf diese Weise war es uns beispielsweise möglich, Benutzereingaben wie die Matrikelnummer automatisch auf ihre Gültigkeit zu überprüfen.
Zum Schluss haben wir uns in diesem Kapitel sehr intensiv mit dem Entstehungsprozess eines Objektes beschäftigt. Wir haben gelernt, wie man mit Konstruktoren dynamische Teile eines Objektes initialisiert und wie man **`static`**-Blöcke einsetzt, um statische Komponenten und Konstanten mit Werten zu belegen. Ferner haben wir uns mit dem Überladen von Konstruktoren befasst und an einem konkreten Beispiel erfahren, wie das Zusammenspiel von Initialisierern und Konstruktoren in Sub- und Superklasse funktioniert.

8.6 Übungsaufgaben

Aufgabe 8.1

Fügen Sie der Klasse `Student` einen weiteren Konstruktor hinzu. In diesem Konstruktor soll man in der Lage sein, alle Instanzvariablen (Name, Nummer, Fach, Geburtsjahr) als Argumente zu übergeben. Erhöhen Sie den Zähler hierbei nicht selbst, sondern verwenden Sie das Schlüsselwort **`this`**, um einen der bereits vorhandenen Konstruktoren aufzurufen. Übergeben Sie diesem Konstruktor auch das gewünschte Geburtsjahr.

Aufgabe 8.2

Fügen Sie der Klasse `Student` eine weitere private Instanzvariable `geschlecht` sowie finale Klassenvariablen `WEIBLICH` und `MAENNLICH` hinzu, sodass beim Arbeiten mit Objekten der Klasse `Student` explizit zwischen weiblichen und männlichen Studierenden unterschieden werden kann. Fügen Sie der Klasse `Student` weitere Konstruktoren hinzu, die diese neuen Variablen berücksichtigen. Verwenden Sie auch hier mit Hilfe des Schlüsselworts **`this`** bereits vorhandene Konstruktoren.

Aufgabe 8.3

Wir nehmen an, dass alle Karlsruher Hochschulen über ein besonderes System verfügen, um Matrikelnummern auf Korrektheit zu überprüfen:

- Zuerst wird die (als siebenstellig festgelegte) Zahl in ihre Ziffern $Z_1, Z_2 \ldots Z_7$ aufgeteilt; für die Matrikelnummer 0848600 wäre also etwa

$$Z_1 = 0,\ Z_2 = 8,\ Z_3 = 4,\ Z_4 = 8,\ Z_5 = 6,\ Z_6 = 0,\ Z_7 = 0.$$

- Nun wird eine spezielle „gewichtete Quersumme“ Σ der Form

$$\Sigma = Z_1 \cdot 2 + Z_2 \cdot 1 + Z_3 \cdot 4 + Z_4 \cdot 3 + Z_5 \cdot 2 + Z_6 \cdot 1$$

  gebildet.
- Die Matrikelnummer ist genau dann gültig, wenn die letzte Ziffer der Matrikelnummer (also Z_7) mit der letzten Ziffer der Quersumme Σ übereinstimmt.

Sie sollen nun eine spezielle Klasse `KarlsruherStudent` entwickeln, die lediglich Zahlen als Matrikelnummern zulässt, die diese Prüfung bestehen. Beginnen Sie zu diesem Zweck mit folgendem Ansatz:

```
1 /** Ein Student einer Karlsruher Hochschule */
2 public class KarlsruherStudent extends Student {
3
4 }
```

Die Klasse leitet sich wegen des Schlüsselworts **`extends`** von unserer allgemeinen Klasse `Student` ab, erbt somit also auch alle Variablen und Methoden. Gehen Sie nun in zwei Schritten vor, um unsere Klasse zu vervollständigen:

a) Im Moment haben wir bei der neuen Klasse nicht die Möglichkeit, das Geburtsjahr zu setzen (machen Sie sich klar, warum). Aus diesem Grund verfassen Sie einen Konstruktor, dem man das Geburtsjahr als Argument übergeben kann. Da Sie keinen Zugriff auf die privaten Instanzvariablen haben, müssen Sie hierzu den entsprechenden Konstruktor der Superklasse aufrufen.

b) Überschreiben Sie die `validateNummer`-Methode so, dass diese die Prüfung gemäß dem Karlsruher System durchführt. Aufgrund des Polymorphismus wird die neue Methode das Original in allen Karlsruher Studentenobjekten ersetzen. Da die set-Methode jedoch die Validierung verwendet, haben wir die Wertzuweisung automatisch dem neuen System angepasst.

   Hinweis: Das Aufspalten einer Zahl in ihre Einzelziffern haben wir in diesem Buch schon an mehreren Stellen besprochen. Verwenden Sie bereits vorhandene Algorithmen, und sparen Sie sich somit den Aufwand einer Neuentwicklung.

Aufgabe 8.4

Vervollständigen Sie den nachfolgenden Lückentext mit Angaben, die sich auf die Klassen `Klang`, `Krach` und `Musik` beziehen, die am Ende dieser Aufgabe angegeben sind:

a) Die Klasse ... ist Superklasse der Klasse ....

b) Die Klasse ... erbt von der Klasse ... die Variable(n) ....

c) In den drei Klassen gibt es die Instanzvariable(n) ....

d) In den drei Klassen gibt es die Klassenvariable(n) ....

e) Auf die Variable(n) ... der Klasse `Klang` kann in der Klasse `Krach` und in der Klasse `Musik` zugegriffen werden.

f) Auf die Variable(n) ... der Klasse `Krach` hat keine andere Klasse Zugriff.

g) Die Variable(n) ... hat/haben in allen Instanzen der Klasse `Krach` den gleichen Wert.

h) Der Konstruktor der Klasse `Klang` wird in den Zeilen ... aufgerufen.

i) Die Methode `mehrPower` der Klasse `Klang` wird in den Zeilen ... bis ... überschrieben und in den Zeilen ... bis ... überladen.

j) Die Methode `mehrPower`, die in den Zeilen ... bis ... definiert ist, wird in Zeile ... und in Zeile ... aufgerufen.

k) Die Methode `mehrPower`, die in den Zeilen ... bis ... definiert ist, wird in Zeile ... aufgerufen.

l) Die Methode `mehrPower`, die in den Zeilen ... bis ... definiert ist, wird in ... aufgerufen.

m) Die Methode `toString`, die in den Zeilen 7 bis 9 definiert ist, wird in ... aufgerufen.

n) Die Methoden ... sind Instanzmethoden.

Auf die nachfolgenden Klassen sollen sich Ihre Antworten beziehen:

```
1 public class Klang {
2   public int baesse, hoehen;
3   public Klang(int b, int h) {
4     baesse = b;
5     hoehen = h;
6   }
7   public String toString () {
8     return "B:" + baesse + " H:" + hoehen;
9   }
```

```
10    public void mehrPower (int b) {
11      baesse += b;
12    }
13  }
14  public class Krach extends Klang {
15    private int rauschen, lautstaerke;
16    public static int grundRauschen = 4;
17    public Krach (int l, int b, int h) {
18      super(b,h);
19      lautstaerke = l;
20      rauschen = grundRauschen;
21    }
22    public void mehrPower (int b) {
23      baesse += b;
24      if (baesse > 10) {
25        lautstaerke -= 1;
26      }
27    }
28    public void mehrPower (int l, int b) {
29      lautstaerke += l;
30      this.mehrPower(b);
31    }
32  }
33  public class Musik {
34    public static void main (String[] args) {
35      Klang k = new Klang(1,5);
36      Krach r = new Krach(4,17,30);
37      System.out.println(r);
38      r.mehrPower(3);
39      r.mehrPower(2,2);
40    }
41  }
```

Aufgabe 8.5

Gegeben seien die folgenden Java-Klassen:

```
1   class Maus {
2     Maus() {
3       System.out.println("Maus");
4     }
5   }
6
7   class Katze {
8     Katze() {
9       System.out.println("Katze");
10    }
11  }
12
13  class Ratte extends Maus {
14    Ratte() {
15      System.out.println("Ratte");
16    }
17  }
18
```

```
19 class Fuchs extends Katze {
20   Fuchs() {
21     System.out.println("Fuchs");
22   }
23 }
24
25 class Hund extends Fuchs {
26   Maus m = new Maus();
27   Ratte r = new Ratte();
28   Hund() {
29     System.out.println("Hund");
30   }
31   public static void main(String[] args) {
32     new Hund();
33   }
34 }
```

Geben Sie an, was beim Start der Klasse `Hund` ausgegeben wird.

Aufgabe 8.6

Gegeben seien die folgenden Klassen:

```
1  class Eins {
2    public long f;
3    public static long g = 2;
4    public Eins (long f) {
5      this.f = f;
6    }
7    public Object clone() {
8      return new Eins(f + g);
9    }
10 }
11
12 class Zwei {
13   public Eins h;
14   public Zwei (Eins eins) {
15     h = eins;
16   }
17   public Object clone() {
18     return new Zwei(h);
19   }
20 }
21
22 public class TestZwei {
23   public static void main (String[] args) {
24     Eins e1 = new Eins(1), e2;
25     Zwei z1 = new Zwei(e1), z2;
26     System.out.print  (Eins.g + "-");
27     System.out.println(z1.h.f);
28     e2 = (Eins) e1.clone();
29     z2 = (Zwei) z1.clone();
30     e1.f = 4;
31     Eins.g = 5;
32     System.out.print  (e2.f + "-");
33     System.out.print  (e2.g + "-");
```

```
34      System.out.print  (z1.h.f + "-");
35      System.out.print  (z2.h.f + "-");
36      System.out.println(z2.h.g);
37    }
38  }
```

Geben Sie an, was beim Aufruf der Klasse `TestZwei` ausgegeben wird.

Aufgabe 8.7

Die folgenden sechs Miniaturprogramme haben alle ein und denselben Sinn. Sie definieren eine Klasse, die eine private Instanzvariable besitzt, die bei der Instantiierung gesetzt werden soll. Mit Hilfe einer `toString`-Methode kann ein derart erzeugtes Objekt (in der `main`-Methode) auf dem Bildschirm ausgegeben werden. Von diesen sechs Programmen sind zwei jedoch dermaßen verkehrt, dass sie beim Übersetzen einen Compilerfehler erzeugen. Drei weitere Programme beinhalten logische Fehler, die der Compiler zwar nicht erkennen kann, die aber bei Ablauf des Programms zutage treten. Finden Sie das eine funktionierende Programm, *ohne* die Programme in den Computer einzugeben. Begründen Sie bei den anderen Programmen jeweils, warum sie nicht funktionieren:

```
1  public class Fehler1 {
2
3    /** Private Instanzvariable */
4    private String name;
5
6    /** Konstruktor */
7    public Fehler1(String name) {
8      name = name;
9    }
10
11   /** String-Ausgabe */
12   public String toString() {
13     return "Name = " + name;
14   }
15
16   /** Hauptprogramm */
17   public static void main(String[] args) {
18     System.out.println(new Fehler1("Testname"));
19   }
20
21 }
```

```
1  public class Fehler2 {
2
3    /** Private Instanzvariable */
4    private String name;
5
6    /** Konstruktor */
7    public Fehler2(String name) {
8      this.name = name;
9    }
10
```

```
11    /** String-Ausgabe */
12    public String tostring() {
13      return "Name = " + name;
14    }
15
16    /** Hauptprogramm */
17    public static void main(String[] args) {
18      System.out.println(new Fehler2("Testname"));
19    }
20
21  }
```

```
1   public class Fehler3 {
2
3     /** Private Instanzvariable */
4     private String name;
5
6     /** Konstruktor */
7     public Fehler3(String nom) {
8       name = nom;
9     }
10
11    /** String-Ausgabe */
12    public String toString() {
13      return "Name = " + name;
14    }
15
16    /** Hauptprogramm */
17    public static void main(String[] args) {
18      System.out.println(new Fehler2("Testname"));
19    }
20
21  }
```

```
1   public class Fehler4 {
2
3     /** Private Instanzvariable */
4     private String name;
5
6     /** Konstruktor */
7     public Fehler4(String nom) {
8       name = nom;
9     }
10
11    /** String-Ausgabe */
12    public String toString() {
13      return "Name = " + name;
14    }
15
16    /** Hauptprogramm */
17    public static void main(String[] args) {
18      System.out.println(new Fehler4("Testname"));
19    }
20
21  }
```

```
1  public class Fehler5 {
2
3    /** Private Instanzvariable */
4    private String name;
5
6    /** Konstruktor */
7    public void Fehler5(String name) {
8      this.name = name;
9    }
10
11   /** String-Ausgabe */
12   public String toString() {
13     return "Name = " + name;
14   }
15
16   /** Hauptprogramm */
17   public static void main(String[] args) {
18     System.out.println(new Fehler5("Testname"));
19   }
20
21 }
```

```
1  public class Fehler6 {
2
3    /** Private Instanzvariable */
4    private String name;
5
6    /** Konstruktor */
7    public Fehler6(String nom) {
8      name = nom;
9    }
10
11   /** String-Ausgabe */
12   public String toString() {
13     return "Name = " + name;
14   }
15
16   /** Hauptprogramm */
17   public static void main(String[] args) {
18     Fehler6 variable = new Fehler6();
19     variable.name = "Testname";
20     System.out.println(variable);
21   }
22
23 }
```

Aufgabe 8.8

Es sei folgende einfache Klasse gegeben, die zur Speicherung von Daten über Tennisspieler (zum Beispiel bei einem Turnier) verwendet werden könnte.

```
1  public class TennisSpieler {
2    public String name;                    // Name des Spielers
3    public int alter;                      // Alter in Jahren
```

```
4     public int altersDifferenz (int alter) {
5       return Math.abs(alter - this.alter);
6     }
7   }
```

a) Erläutern Sie den Aufbau der Klasse grafisch.

b) Was passiert durch die nachfolgenden Anweisungen?

```
TennisSpieler maier;
maier = new TennisSpieler();
```

Warum benötigt man die zweite Anweisung überhaupt?

c) Erläutern Sie die Bedeutung der `this`-Referenz grafisch und anhand der Methode `altersDifferenz`.

d) Wie erfolgt der Zugriff auf die Daten (Variablen) und Methoden der Klasse?

e) Was versteht man unter einem Konstruktor, und wie würde ein geeigneter Konstruktor für die Klasse `TennisSpieler` aussehen? Wenn Sie die Klasse um diesen Konstruktor ergänzen, ist dann die Anweisung

```
TennisSpieler maier = new TennisSpieler();
```

noch zulässig?

f) Erläutern Sie den Unterschied zwischen Instanzvariablen und Klassenvariablen.

g) Erweitern Sie die Klasse `TennisSpieler` um eine Instanzvariable namens `verfolger`, die eine Referenz auf einen weiteren Tennisspieler (den unmittelbaren Verfolger in der Weltrangliste) darstellt, und um eine Instanzvariable `startNummer`, die es ermöglicht, allen Tennisspielern (z. B. bei der Erzeugung eines neuen Objektes für eine Teilnehmerliste eines Turniers) eine (eindeutige) ganzzahlige Nummer zuzuordnen.

h) Erweitern Sie die Klasse `TennisSpieler` um eine Klassenvariable namens `folgeNummer`, die die jeweils nächste zu vergebende Startnummer enthält.

i) Modifizieren Sie den Konstruktor der Klasse `TennisSpieler` so, dass er jeweils eine entsprechende Startnummer vergibt und die Klassenvariable `folgeNummer` jeweils erhöht. Geben Sie auch eine Überladung dieses Konstruktors an, die es ermöglicht, bei der Objekterzeugung auch noch den Verfolger in der Weltrangliste anzugeben.

j) Wie verändert sich der Wert der Variablen `startNummer` und `folgeNummer` in den Objekten `maier`, `schmid` und `berger` mit den nachfolgenden Anweisungen?

```
TennisSpieler maier  = new TennisSpieler("H. Maier", 68);
TennisSpieler schmid = new TennisSpieler("G. Schmid", 45, maier);
TennisSpieler berger = new TennisSpieler("I. Berger", 36, schmid);
```

k) Erläutern Sie den Unterschied zwischen Instanzmethoden und Klassenmethoden.

l) Erweitern Sie die Klasse `TennisSpieler` um eine Instanzmethode namens `istLetzter`, die genau dann den Wert **`true`** liefert, wenn das Tennisspieler-Objekt keinen Verfolger in der Weltrangliste hat.

m) Erweitern Sie die Klasse `TennisSpieler` um die Instanzmethode

```
public String toString () {
  String printText = name + " (" + startNummer + ")";
  if (verfolger != null)
    printText = printText + " liegt vor " + verfolger.toString();
  return printText;
}
```

die es ermöglicht, dass man Objekte der Klasse innerhalb von Zeichenkettenausdrücken (also auch in Ausgabeanweisungen) mit + verknüpfen bzw. automatisch nach String wandeln lassen kann. Was würden die Zeilen

```
System.out.println(maier);
System.out.println(schmid);
System.out.println(berger);
```

ausgeben?

n) Wie kann man vermeiden, dass ein(e) Programmierer(in) bei der Bearbeitung der Objekte der Klasse `TennisSpieler` die (von den Konstruktoren automatisch generierten) Startnummern überschreibt? Wie lässt sich dann trotzdem lesender Zugriff auf die Startnummern ermöglichen?

Aufgabe 8.9

Schreiben Sie eine Klasse `Mensch`, die *private* Instanzvariablen beinhaltet, um eine laufende Nummer (**`int`**), den Vornamen (`String`), den Nachnamen (`String`), das Alter (**`int`**) und das Geschlecht (**`boolean`**, mit **`true`** für männlich) eines Menschen zu speichern. Außerdem soll die Klasse eine private Klassenvariable namens `gesamtZahl` (zur Information über die Anzahl der bereits erzeugten Objekte der Klasse) beinhalten, die mit dem Wert 0 zu initialisieren ist.
Statten Sie die Klasse mit einem Konstruktor aus, der als Parameter das Alter als **`int`**-Wert, das Geschlecht als **`boolean`**-Wert und den Vor- und Nachnamen als `String`-Werte übergeben bekommt und die entsprechenden Instanzvariablen des Objekts mit diesen Werten belegt. Außerdem soll der Objektzähler `gesamtZahl` um 1 erhöht und danach die laufende Nummer des Objekts auf den neuen Wert von `gesamtZahl` gesetzt werden.
Statten Sie die Klasse außerdem mit folgenden Instanz-Methoden aus:

a) **`public int`** `getAlter()`
Diese Methode soll das Alter des Objekts zurückliefern.

b) **public void** `setAlter(`**int** `neuesAlter)`
Diese Methode soll das Alter des Objekts auf den Wert `neuesAlter` setzen.

c) **public boolean** `getIstMaennlich()`
Diese Methode soll den **boolean**-Wert (die Angabe des Geschlechts) des Objekts zurückliefern.

d) **public boolean** `aelterAls(Mensch m)`
Wenn das Alter des Objekts größer ist als das Alter von `m`, soll diese Methode den Wert **true** zurückliefern, andernfalls den Wert **false**.

e) **public** `String toString()` Diese Methode soll eine Zeichenkette zurückliefern, die sich aus dem Vornamen, dem Nachnamen, dem Alter, dem Geschlecht und der laufenden Nummer des Objekts zusammensetzt.

Zum Test Ihrer Klasse `Mensch` können Sie eine einfache Klasse `TestMensch` schreiben, die mit Objekten der Klasse `Mensch` arbeitet und den Konstruktor und alle Methoden der Klasse `Mensch` testet. Testen Sie dabei auch,

- ob der Compiler wirklich Zugriffe auf die privaten Instanzvariablen verweigert und
- ob der Compiler für ein Objekt `m` der Klasse `Mensch` tatsächlich bei einer Anweisung

```
System.out.println(m);
```

automatisch die `toString()`-Methode aufruft!

Aufgabe 8.10

Ein Punkt p in der Ebene mit der Darstellung $p = (x_p, y_p)$ besitzt die x-Koordinate x_p und die y-Koordinate y_p. Die Strecke $\overline{pq}$ zwischen zwei Punkten $p = (x_p, y_p)$ und $q = (x_q, y_q)$ hat nach Pythagoras die Länge $L(\overline{pq}) = \sqrt{(x_q - x_p)^2 + (y_q - y_p)^2}$ (siehe auch Abbildung 8.8).

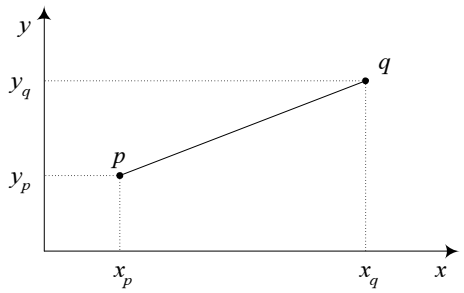

Abbildung 8.8: Definition einer Strecke

Unter Verwendung der objektorientierten Konzepte von Java soll in einem Programm mit solchen Punkten und Strecken in der Ebene gearbeitet werden. Dazu sollen

- eine Klasse `Punkt` zur Darstellung und Bearbeitung von Punkten,
- eine Klasse `Strecke` zur Darstellung und Bearbeitung von Strecken und
- eine Klasse `TestStrecke` für den Test bzw. die Anwendung dieser beiden Klassen

implementiert werden. Gehen Sie wie folgt vor.

a) Implementieren Sie die Klasse `Punkt` mit zwei privaten Instanzvariablen `x` und `y` vom Typ **`double`**, die die x- und y-Koordinaten eines Punktes repräsentieren, und statten Sie die Klasse `Punkt` mit Konstruktoren und Instanzmethoden aus. Schreiben Sie

   - einen Konstruktor mit zwei **`double`**-Parametern (die x- und y-Koordinaten des Punktes),
   - eine Methode `getX()`, die die x-Koordinate des Objekts zurückliefert,
   - eine Methode `getY()`, die die y-Koordinate des Objekts zurückliefert,
   - eine **`void`**-Methode `read()`, die die x- und y-Koordinaten des Objekts einliest, und
   - eine `String`-Methode `toString()`, die die `String`-Darstellung des Objekts in der Form `(xStr,yStr)` zurückliefert, wobei `xStr` und `yStr` die `String`-Darstellungen der Werte von `x` und `y` sind.

b) Implementieren Sie die Klasse `Strecke` mit zwei privaten Instanzvariablen `p` und `q` vom Typ `Punkt`, die die beiden Randpunkte einer Strecke repräsentieren, und statten Sie die Klasse `Strecke` mit Konstruktoren und Instanzmethoden aus. Schreiben Sie

   - einen Konstruktor mit zwei `Punkt`-Parametern (die Randpunkte der Strecke),
   - eine **`void`**-Methode `read()`, die die beiden Randpunkte `p` und `q` des Objekts einliest (verwenden Sie dazu die Instanzmethode `read` der Objekte `p` und `q`),
   - eine **`double`**-Methode `getLaenge()`, die (unter Verwendung der Instanzmethoden `getX` und `getY` der Randpunkte) die Länge des Strecken-Objekts berechnet und zurückliefert,
   - eine `String`-Methode `toString()`, die die `String`-Darstellung des Objekts in der Form `pStr_qStr` zurückliefert, wobei `pStr` und `qStr` die `String`-Darstellungen für die Instanzvariablen p und q des Objekts sind.

c) Testen Sie Ihre Implementierung mit der folgenden Klasse:

```
1  public class TestStrecke {
2    public static void main(String[] args) {
3      Punkt ursprung = new Punkt(0.0,0.0);
4      Punkt endpunkt = new Punkt(4.0,3.0);
5      Strecke s = new Strecke(ursprung,endpunkt);
6      System.out.println("Die Laenge der Strecke " + s +
7                         " betraegt " + s.getLaenge() + ".");
8      System.out.println();
9      System.out.println("Strecke s eingeben:");
10     s.read();
11     System.out.println();
12     System.out.println("Die Laenge der Strecke " + s +
13                        " betraegt " + s.getLaenge() + ".");
14   }
15 }
```

Aufgabe 8.11

Gegeben sei die folgende Klasse:

```
1  public class AchJa {
2
3    public int x;
4    static int ach;
5
6    int ja (int i, int j) {
7      int y;
8      if ((i <= 0) || (j <= 0) || (i % j == 0) || (j % i == 0)) {
9         System.out.print(i+j);
10        return i + j;
11     }
12     else {
13        x = ja(i-2,j);
14        System.out.print(" + ");
15        y = ja(i,j-2);
16        return x + y;
17     }
18   }
19
20   public static void main (String[] args) {
21     int n = 5, k = 2;
22     AchJa so = new AchJa();
23     System.out.print("ja(" + n + "," + k + ") = ");
24     ach = so.ja(n,k);
25     System.out.println(" = " + ach);
26   }
27 }
```

a) Geben Sie an, um welche Art von Variablen es sich bei den in dieser Klasse verwendeten Variablen `x` in Zeile 2, `ach` in Zeile 3, `j` in Zeile 4, `y` in Zeile 5, `n` in Zeile 18 und `so` in Zeile 19 jeweils handelt. Verwenden Sie (sofern diese zutreffen) die Bezeichnungen Klassen-Variable, Instanz-Variable, lokale Variable und formale Variable (bzw. formaler Parameter).

b) Geben Sie an, was das Programm ausgibt.

c) Angenommen, die Zeile 21 würde in der Form

```
ach = ja(n,k);
```

gegeben sein. Würde der Compiler das Programm trotzdem übersetzen? Wenn nein, warum nicht?

Aufgabe 8.12

Es sei folgende einfache Klasse gegeben, die zur Speicherung von Daten über Patienten in der Aufnahme einer Arztpraxis verwendet werden könnte.

```
1 public class Patient {
2   public String name;          // Name des Patienten
3   public int alter;            // Alter (in Jahren)
4   public int altersDifferenz (int alter) {
5     return Math.abs(alter - this.alter);
6   }
7 }
```

a) Erläutern Sie den Aufbau der Klasse grafisch.

b) Was passiert durch die nachfolgenden Anweisungen?

```
Patient maier;
maier = new Patient();
```

c) Wie würde ein geeigneter Konstruktor für die Klasse `Patient` aussehen? Wenn Sie die Klasse um diesen Konstruktor ergänzen, ist dann die Anweisung

```
Patient maier = new Patient();
```

noch zulässig?

d) Erweitern Sie die Klasse `Patient` um eine Instanzvariable `vorherDran`, die eine Referenz auf einen weiteren Patienten darstellt, und um eine Instanzvariable `nummer`, die es ermöglicht, allen Patienten (z. B. bei der Erzeugung eines neuen Objektes für eine Warteliste einer Praxis) eine (eindeutige) ganzzahlige Nummer zuzuordnen.

e) Erweitern Sie die Klasse `Patient` um eine Klassenvariable `folgeNummer`, die die jeweils nächste zu vergebende Nummer enthält.

f) Modifizieren Sie den Konstruktor der Klasse `Patient` so, dass er jeweils eine entsprechende Nummer vergibt und die Klassenvariable `folgeNummer` jeweils erhöht. Geben Sie auch eine Überladung dieses Konstruktors an, die es ermöglicht, auch noch den Vorgänger in der Warteliste anzugeben.

g) Wie verändert sich der Wert der Variablen `nummer` und `folgeNummer` in den Objekten `maier`, `schmid` und `berger` mit den nachfolgenden Anweisungen?

```
Patient maier  = new Patient("H. Maier", 68);
Patient schmid = new Patient("G. Schmid", 45, maier);
Patient berger = new Patient("I. Berger", 36, schmid);
```

h) Erweitern Sie die Klasse `Patient` um eine Instanzmethode `istErster`, die genau dann den Wert **`true`** liefert, wenn das Patienten-Objekt keinen Vorgänger in der Warteliste hat.

i) Erweitern Sie die Klasse `Patient` um die Instanzmethode

```
public String toString () {
  String printText = name + " (" + nummer + ")";
  if (vorherDran != null)
    printText = printText + " kommt nach " + vorherDran.toString();
  return printText;
}
```

die es ermöglicht, dass man Objekte der Klasse innerhalb von Zeichenkettenausdrücken (also auch in Ausgabeanweisungen) mit + verknüpfen kann. Was würden die Zeilen

```
System.out.println(maier);
System.out.println(schmid);
System.out.println(berger);
```

ausgeben?

j) Wie vermeidet man, dass ein(e) Programmierer(in) bei der Bearbeitung der Objekte der Klasse `Patient` die (von den Konstruktoren automatisch generierten) Nummern überschreibt? Wie ermöglicht man dann trotzdem lesenden Zugriff auf die Identifikationsnummern?

Aufgabe 8.13

Sie sollen verschiedene Fahrzeuge mittels objektorientierter Programmierung simulieren. Dazu ist Ihnen folgende Klasse vorgegeben:

```
1  public class Reifen {
2
3    /** Reifendruck */
4    private double druck;
5
6    /** Konstruktor */
7    public Reifen (double luftdruck) {
8      druck = luftdruck;
9    }
10
11   /** Zugriffsfunktion fuer Reifendruck */
12   public double aktuellerDruck () {
13     return druck;
14   }
15 }
```

Schreiben Sie eine Klasse `Fahrzeug`, die die Klasse `Reifen` verwendet und Folgendes beinhaltet:

a) **private** Instanzvariablen

- `name` vom Typ `String` (für die Bezeichnung des Fahrzeugs),
- `anzahlReifen` vom Typ **`int`** (für die Anzahl der Reifen des Fahrzeugs),
- `reifenArt` vom Typ `Reifen` (für die Angabe des Reifentyps des Fahrzeugs) und
- `faehrt` vom Typ **`boolean`** (für die Information über den Fahrzustand des Fahrzeugs);

b) einen Konstruktor, der mit Parametern für Bezeichnung, Reifenanzahl und Reifendruck ausgestattet ist, in seinem Rumpf die entsprechenden Komponenten des Objekts belegt und außerdem das Fahrzeug in den Zustand „fährt nicht“ versetzt;

c) eine öffentliche Instanzmethode `fahreLos()`, die die Variable `faehrt` des `Fahrzeug`-Objektes auf **`true`** setzt;

d) eine öffentliche Instanzmethode `halteAn()`, die die Variable `faehrt` des `Fahrzeug`-Objektes auf **`false`** setzt;

e) eine öffentliche Instanzmethode `status()`, die einen Informations-String über Bezeichnung, Fahrzustand, Reifenzahl und Reifendruck des `Fahrzeug`-Objektes auf den Bildschirm ausgibt.

Aufgabe 8.14

Schreiben Sie ein Testprogramm, das in seiner `main`-Methode zunächst ein Fahrrad (verwenden Sie Reifen mit 4.5 bar) und ein Auto (verwenden Sie Reifen mit 1.9 bar) in Form von Objekten der Klasse `Fahrzeug` erzeugt und anschließend folgende Vorgänge durchführt:

1. mit dem Fahrrad losfahren,
2. mit dem Auto losfahren,
3. mit dem Fahrrad anhalten,
4. mit dem Auto anhalten.

Unmittelbar nach jedem der vier Vorgänge soll jeweils mittels der Methode `status()` der aktuelle Fahrzustand *beider* Fahrzeuge ausgegeben werden.
Eine Ausgabe des Testprogramms sollte also etwa so aussehen:

Konsole

```
Zustand 1:
Fahrrad1 faehrt auf 2 Reifen mit je 4.5 bar
Auto1 steht auf 4 Reifen mit je 1.9 bar
Zustand 2:
```

```
Fahrrad1 faehrt auf 2 Reifen mit je 4.5 bar
Auto1 faehrt auf 4 Reifen mit je 1.9 bar
Zustand 3:
Fahrrad1 steht auf 2 Reifen mit je 4.5 bar
Auto1 faehrt auf 4 Reifen mit je 1.9 bar
Zustand 4:
Fahrrad1 steht auf 2 Reifen mit je 4.5 bar
Auto1 steht auf 4 Reifen mit je 1.9 bar
```

Aufgabe 8.15

Gegeben seien die folgenden Klassen:

```
1  public class IntKlasse {
2    public int a;
3    public IntKlasse (int a) {
4      this.a = a;
5    }
6  }
7  public class RefIntKlasse {
8    public IntKlasse x;
9    public double y;
10   public RefIntKlasse(int u, int v) {
11     x = new IntKlasse(u);
12     y = v;
13   }
14 }
15 public class KlassenTest {
16   public static void copy1 (RefIntKlasse f, RefIntKlasse g) {
17     g.x.a = f.x.a;
18     g.y   = f.y;
19   }
20   public static void copy2 (RefIntKlasse f, RefIntKlasse g) {
21     g.x = f.x;
22     g.y = f.y;
23   }
24   public static void copy3 (RefIntKlasse f, RefIntKlasse g) {
25     g = f;
26   }
27   public static void main (String args[]) {
28     RefIntKlasse p = new RefIntKlasse(5,7);
29     RefIntKlasse q = new RefIntKlasse(1,2); // Ergibt das Ausgangsbild
30     // HIER FOLGT NUN EINE KOPIERAKTION:
31     ... //***
32   }
33 }
```

Das Ausgangsbild (mit Referenzen und Werten), das sich zur Laufzeit unmittelbar vor der Kopieraktion ergibt, sieht wie in Abbildung 8.9 beschrieben aus.

a) Welches Bild würde sich ergeben, wenn unmittelbar vor `//***`

```
copy1(p,q);
```

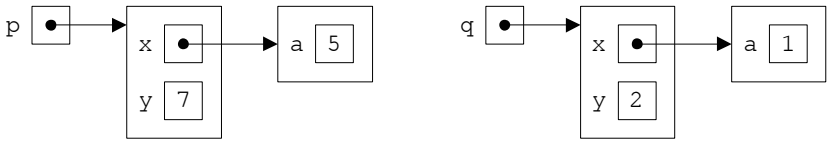

Abbildung 8.9: Ausgangsbild Aufgabe 8.15

stehen würde? Zeichnen Sie den Zustand inklusive der Referenzen und Werte nach der Kopieraktion.

b) Welches Bild würde sich ergeben, wenn unmittelbar vor `//***`

```
copy2(p,q);
```

stehen würde? Zeichnen Sie die Referenzen und Werte nach der Kopieraktion.

c) Welches Bild würde sich ergeben, wenn unmittelbar vor `//***`

```
copy3(p,q);
```

stehen würde? Zeichnen Sie die Referenzen und Werte nach der Kopieraktion.

d) Welches Bild würde sich ergeben, wenn unmittelbar vor `//***`

```
q = p;
```

stehen würde? Zeichnen Sie die Referenzen und Werte nach der Kopieraktion.

Aufgabe 8.16

Gegeben sei folgende Klasse zur Darstellung und Bearbeitung von runden Glasböden:

```
1  public class Glasboden {
2    private double radius;
3    public Glasboden (double r) {
4      radius = r;
5    }
6    public void verkleinern (double x) {
7      // verkleinert den Radius des Glasboden-Objekts um x
8      radius  = radius - x;
9    }
10   public double flaeche () {
11     // liefert die Flaeche des Glasboden-Objekts
12     return Math.PI * radius * radius;
13   }
14   public double umfang () {
15     // liefert den Umfang der Glasboden-Objekts
16     return 2 * Math.PI * radius;
17   }
18   public String toString() {
19     // liefert die String-Darstellung des Glasboden-Objekts
```

```
20      return "B(r=" + radius + ")";
21    }
22  }
```

a) Ergänzen Sie die fehlenden Teile der Klasse `TrinkGlas`, die ein Trinkglas durch jeweils einen Glasboden und durch eine Füllstands-Angabe darstellt:

- Ergänzen Sie zwei private Instanzvariablen `boden` vom Typ `Glasboden` und `fuellStand` vom Typ **`double`** (der Boden und der Füllstand des Glases).
- Vervollständigen Sie den Konstruktor.
- Vervollständigen Sie die Methode `verkleinern`, die die Größe des `TrinkGlas`-Objekts verändert, indem der Glasboden um den Wert `x` verkleinert und der Füllstand des Glases um den Wert `x` verringert wird.
- Vervollständigen Sie die Methode `flaeche()`, die die Innenfläche (siehe Hinweis) des `TrinkGlas`-Objekts berechnet und zurückliefert.
- Vervollständigen Sie die Methode `fuellMenge()`, die die Füllmenge (siehe Hinweis) des `TrinkGlas`-Objekts berechnet und zurückliefert.
- Vervollständigen Sie die Methode `toString()`, die die `String`-Darstellung des Objekts in der Form `G(xxx,s=yyy)` zurückliefert, wobei `xxx` für die `String`-Darstellung der Instanzvariable `boden` und `yyy` für den Wert des Füllstandes des Trinkglases stehen sollen.

Hinweis: Bezeichnen F die Glasboden-Fläche, U den Glasboden-Umfang und s den Füllstand eines Trinkglases, so sollen die Innnenfläche I und die Füllmenge M dieses Trinkglases durch

$$I = F + U \cdot s \qquad \text{und} \qquad M = F \cdot s$$

berechnet werden.

```
public class TrinkGlas {
  .
  .
  .
  .
  .
  public TrinkGlas (double fuellStand, Glasboden boden) {
    .
    .
    .
    .
  }
  public void verkleinern (double x) {
    .
    .
    .
    .
  }
```

```
  public double flaeche() {
    .
    .
    .
    .
  }
  public double fuellMenge() {
    .
    .
    .
    .
  }
  public String toString() {
    .
    .
    .
    .
  }
}
```

b) Ergänzen Sie die nachfolgende Klasse `TesteTrinkGlas`. In deren `main`-Methode soll zunächst ein Trinkglas `g` aus einem Glasboden `b` mit Radius 100 und Füllstand 50 konstruiert werden. Danach soll in einer Schleife das Trinkglas jeweils um den Wert 5 verkleinert und das aktuelle Trinkglas, seine bedeckte Innenfläche und seine Füllmenge ausgegeben werden.

Die Schleife soll nur durchlaufen werden, falls bzw. solange für die Innenfläche I und die Füllmenge M des Trinkglases gilt

$$I < \frac{M}{8}.$$

```
public class TesteTrinkGlas {
  public static void main(String[] args) {
    .
    .
    .
    .
    .
    .
    .
    .
    .
    .
    .
    .
    .
    .
    .
  }
}
```