Klassen in Java

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

In diesem Kapitel werden wir sehen, wie man in Java eigene Klassentypen implementieren kann. Wir stellen außerdem eine Fallstudie vor, die den Umgang mit Klassen und Objekten an einem größeren Beispiel zeigt. Dabei erläutern wir die Komposition von Klassen und Objektgeflechte. Wir implementieren auch als erste klassische Datenstruktur einfachverkettete Listen.

1 Klassendeklarationen in Java

Eine einfache ***Klassendeklaration*** in Java hat folgende Bestandteile:

```
class Person {
  String name;
  Person(String n) {
    this.name = n;
  }
  String getName() {
    return this.name;
  }
}
```

Klassenname

Attribut

Konstruktor

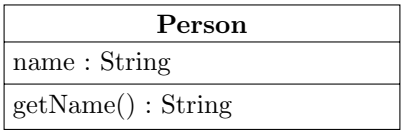
Methode

Ein Java-Objekt kann genau auf die Nachrichten reagieren, für die Methoden in seiner Klasse deklariert sind oder für die es Methoden geerbt hat (⇒ Abschnitt "Vererbung").

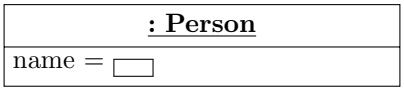
- In der Regel speichern wir den Code für die Deklaration einer Klasse in einer gleichnamigen `.java` - Datei ab.
  Im Beispiel: `Person.java`
- Klassennamen beginnen in Java nach Konvention mit einem Großbuchstaben.

Eine Klasse kann durch ein ***Klassendiagramm*** spezifiziert werden. Klassendiagramme dienen hauptsächlich der Datenmodellierung. Sie sind im UML (Unified Modeling Language) Standard definiert. Sie enthalten den *Name* der Klasse, *Attribute* mit Typ und *Methoden* mit Signaturen.

Beispiel für die `Person`-Klasse:



Die Objekte einer Klasse K nennt man auch ***Instanzen*** oder *Ausprägungen* von K. Sie werden durch Objektdiagramme modelliert, z.B.:



1.1 Klassenname

Direkt hinter dem Schlüsselwort `class` wird der Name der Klasse angegeben.
Der Klassenname wird gleichzeitig als *Typname* für die Objekte dieser Klasse verwendet (*Klassentyp*). Er kann im Programm dann wie elementare Typen (`int`, `double`, usw.) für die Deklaration von lokalen Variablen, Parametern und Rückgabewerten verwendet werden.

1.2 Attribute

Innerhalb einer Klasse K können Attribute deklariert werden.
Für jedes in K deklarierte Attribut vom Typ T besitzen die Objekte der Klasse K eine ***objektlokale*** Variable vom Typ T. Diese objektlokalen Variablen nennt man häufig auch ***Instanzvariablen***.
Die Lebensdauer der Instanzvariablen entspricht der Lebensdauer des Objekts.

1.3 Methoden

Innerhalb der Klassendeklaration können beliebig viele *Methoden* deklariert werden.
Methodendeklarationen bestehen aus einer Signatur und einem Methodenrumpf. Syntaktisch sind sie wie Prozedurdeklarationen aufgebaut.
Außer den deklarierten Parametern besitzt jede Methode m einen weiteren, sogenannten ***impliziten Parameter*** vom Typ der Klasse, in der m deklariert wurde. Dieser Parameter wird im Methodenrumpf mit `this` bezeichnet.
Beispiel:

```
String getName() {
  return this.name;
}
```

Neben den prozeduralen Anweisungen kann eine Methodenrumpf in Java:

- neue Objekte erzeugen,

- auf Attribute zugreifen,
- Nachrichten an andere Objekte schicken (Methodenaufruf).

1.4 Konstruktoren und Initialisierung

Konstruktoren erzeugen und initialisieren Objekte. Sie haben den gleichen Namen wie die Klasse, in der sie deklariert sind. Beim Start der Ausführung eines Konstruktors ist das zugehörige Objekt bereits erzeugt, seine Attribute jedoch nur mit Standardwerten initialisiert. Im Konstruktor werden die Objektattribute geeignet initialisiert, basierend auf den aktuellen Parametern beim Konstruktoraufruf.
Konstruktoren liefern als Ergebnis das neu erzeugte Objekt zurück, genauer: eine Referenz auf dieses Objekt.
Attribute (wie auch lokale Variablen) können alternativ direkt an ihrer Deklarationsstelle initialisiert werden.

```
class C {
  int a = 78;
  C() {}
}
```

```
class C {
  int a;
  C() {
    a = 78;
  }
}
```

Die Initialisierung von Attributen erfolgt vor dem Eintritt in den Konstruktor*rumpf*.
In Java können Attribute und Variablen durch das Schlüsselwort `final` als ***unveränderlich*** deklariert werden. In diesem Fall *muss* die Initialisierung an der Deklarationsstelle erfolgen oder in geeigneter Weise im Konstruktor.

```
class Mathe {
   ...
   final float PI = 3.141; // Konstante
   ...
}
```

1.5 Attributzugriff

Syntax in Java:
Auf Instanzvariablen von Objekten kann mit Ausdrücken folgender Form zugegriffen werden:

Ausdruck → Ausdruck . ≪ *Bezeichner* ≫

Semantik:
Werte den Ausdruck aus; dieser muss eine Referenz liefern.
Liefert dieser `null`, löse eine `NullPointerException` aus.
Andernfalls liefert er die Referenz auf ein Objekt X; in dem Fall liefert der gesamte Ausdruck die Instanzvariable von X zum angegenen Attribut (L-Wert) oder deren Wert (R-Wert).

Abkürzende Notation:
Der implizite Methodenparameter `this` kann beim Zugriff auf ein Attribut `a` weggelassen werden, d.h.

```
a
```

ist gleichbedeutend mit

```
this.a
```

innerhalb von Klassen, in denen `a` deklariert ist.
Beispiel: Attributzugriffe

```
class Autor {
  String name;
  int    geburtsjahr;
}


class Buch {
  Autor  autor;
  String titel;

   void printInfo() {
      StdOut.println("Titel:" + this.titel);
      StdOut.println("Autor:" + this.autor.name);
   }
}
```

Wie beim Attributzugriff kann auch beim Methodenaufruf *innerhalb der Klassendeklaration* der implizite Methodenparameter `this` weggelassen werden, also `m(...)` statt `this.m(...)`.
Hier ein weiteres Beispiel:

```
class Mensch {
  Mensch vater, mutter;
  String name;

  Mensch getOpa (boolean mutterseits) {
    if (mutterseits) {
      return mutter.vater;
    } else {
      return vater.vater;
    }
  }
```

```
  void ermittleOpa(Mensch m) {
     Mensch opaV;
     String opaMName;
     opaV     = m.getOpa(false);
     opaMName = m.getOpa(true).name;
  }
}
```

2 Fallstudie: Lauftagebuch

In diesem Abschnitt zeigen wir das Erstellen von Klassen in einem größeren Kontext. Hier ist die Problembeschreibung:

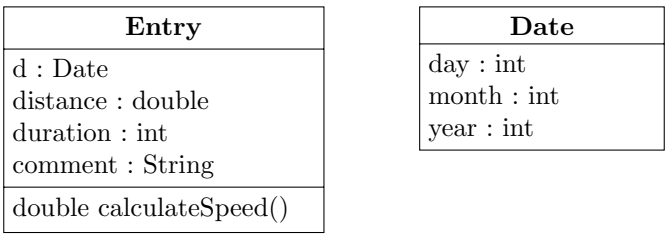
> Entwickle ein Programm, das ein persönliches Lauftagebuch führt. Es enthält einen Eintrag pro Lauf. Einträge können hinzugefügt und entfernt werden. Außerdem soll der Eintrag an einer bestimmten Position ermittelt werden.
>
> Ein Eintrag besteht aus dem Datum, der zurückgelegten Entfernung, der Dauer des Laufs und einem Kommentar zum Zustand des Läufers nach dem Lauf. Für einen Eintrag kann man die Durchschnittsgeschwindigkeit ermitteln.
>
> Ein Datum besteht aus Tag, Monat und Jahr.

Um ein Programm für diese Aufgabenstellung zu implementieren, gehen wir wie folgt vor:

1. Studiere die Problembeschreibung. Identifiziere die darin beschriebenen Objekte und ihre Attribute und Methoden.
2. Erstelle entsprechende Klassendiagramme. (*Entwurf*)
3. Übersetze die Klassendiagramm in eine Klassendefinition. Füge einen Kommentar hinzu, der den Zweck der Klasse erklärt. (*Implementierung*)
4. Repräsentiere einige Beispiele durch Objekte. Erstelle Objekte und stelle fest, ob sie Beispielobjekten entsprechen. (*Test*)

- Substantive in der Beschreibung liefern Hinweise auf Klassen oder Attribute.
- Verben liefern Hinweise für Methoden.

2.1 Klassendiagramm



Aus dem Klassendiagramm lässt sich sehr einfach ein Gerüst für die Implementierung ableiten. Die Implementierung der Methoen lasse wir dabei zunächst offen.

```
// Eintrag in einem Lauftagebuch
class Entry {
    Date d;
    double distance; // in km
    int duration;    // in min
    String comment
    Entry(Date d, double distance, int duration,
          String comment) {
        this.d = d;
        this.distance = distance;
        this.duration = duration;
        this.comment = comment;
    }

    // Ermittelt die Durchschnittgeschwindigkeit
    double calcuateSpeed() {
        //TODO
        return 0.0;
    }
}

// Datum mit Tag, Monat, Jahr
class Date {
    int day;
    int month;
    int year;

    Date(int day, int month, int year) {
      this.day = day;
      this.month = month;
      this.year = year;
    }
}
```

Wir wollen die folgenden Einträge im Lauftagebuch eintragen:

- am 5. Juni 2015, 8.5 km in 27 Minuten, gut
- am 6. Juni 2015, 4.5 km in 24 Minuten, müde
- am 23. Juni 2015, 42.2 km in 150 Minuten, erschöpft

Dazu erzeugen wir die folgenden Objekte:

```
new Entry (new Date (5,6,2015), 8.5, 27, "gut")
new Entry (new Date (6,6,2015), 4.5, 24, "muede")
new Entry (new Date (23,6,2015), 42.2, 150, "k.o.")
```

Diese kompakte Konstuktorverschachtelung kann auch in zwei Schritten mit Hilfsdefinitionen erfolgen:

```
Date d1 = new Date (5,6,2015);
Entry e1 = new Entry (d1, 8.5, 27, "gut");
```

Wir können die `Entry`-Klasse um die Berechnung der durschnittlichen Geschwindigkeit erweitern:

```
class Entry {
    Date d;
    double distance; // in km
    int duration;    // in min
    String comment;
    ...
    double calculateSpeed() {
        double h = duration / 60.0;
        return distance / h;
    }
}
```

Außerdem fügen wir der `Date`-Klasse eine Methode für die Darstellung als String hinzu:

```
class Date {
    int day;
    int month;
    int year;
    ...
    public String toString() {
        return day + "." + month + "." + year;
    }
}
```

Alle Referenztypen in Java haben eine Methode `toString()`. Man kann eigene String-Repräsentationen definieren. Diese müssen als `public` deklariert werden (⇒ Abschnitt "Vererbung"). Die String-Repräsentation enthält üblicherweise Informationen zu den Attributwerten. `toString()` wird (automatisch) aufgerufen, wenn das Objekt in einem Kontext verwendet wird, das einen String erwartet.

```
Date d = new Date (5,6,2015);
StdOut.println(d.toString());
StdOut.println(d); //alternativ
```

Für die Implementierung der `toString()` ***delegieren*** wir die String-Repräsentierung des Datumsattributs an die `Date`-Klasse.

```
class Entry {
  Date d;
  double distance; // in km
  int duration;    // in min
  String comment;
  ...
  public String toString() {
        return d.toString() + ": " + distance + " km in "
            + duration + " min; " + comment;
```

```
  }
}
```

Dieses Muster ist typisch für Klassen, deren Objekte (u.a.) aus Objekten anderer Klassen zusammengesetzt sind (sogenannte ***Aggregate*** oder ***Kompositionen***).

2.2 Objektgeflechte

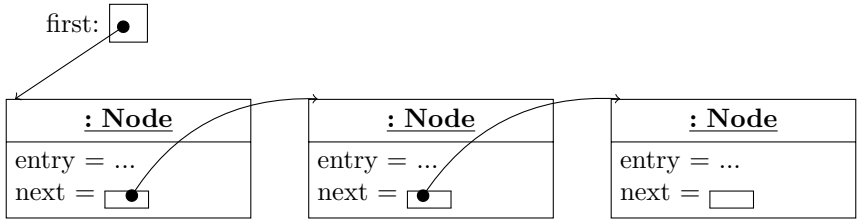
Unsere Implementierung umfasst eine Klasse `Entry` für einzelne Einträge im Tagebuch. Noch offen ist die Problemstellung ein Tagebuch mit **beliebiger Anzahl** von `Entry` -Objekten zu modellieren und implementieren. Ein Möglichkeit ist es das Tagebuch durch eine ***Liste*** von Einträgen zu repräsentieren. Dabei soll jeder Eintrag auf den nächsten Eintrag in der Liste verweisen.
Eine Menge von Objekten, die sich gegenseitig referenzieren, nennen wir ein ***Objektgeflecht***. Objektgeflechte werden zur Laufzeit aufgebaut und verändert, sind also dynamische Entitäten. Klassendiagramme kann man als vereinfachte statische Approximationen von Objektgeflechten verstehen.

2.3 Einfachverkettete Listen

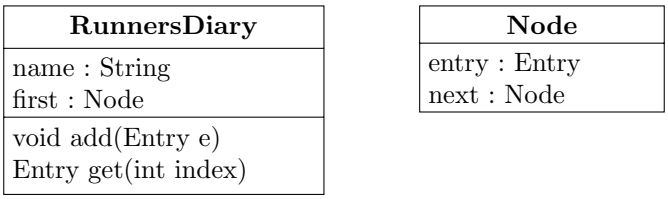
Bei einfachverketteten Listen wird für jedes Listenelement ein Objekt mit zwei Attributen angelegt:

- zum Speichern des Elements
- zum Speichern der Referenz auf den Rest der Liste.

Die Liste erhält also folgende Repräsentation:



Für das Lauftagebuch ergibt sich damit folgendes Klassendiagramm:



Wir können daraus ein Grundgerüst für die Implementierung ableiten:

```
class RunnersDiary {
    String name;
    Node first;
    RunnersDiary(String name) {
        this.name = name;
    }
    // Fuegt einen neuen Eintrag hinzu
    void add(Entry e) {
      // TODO
    }
    // Entfernt alle Eintraege zu einem Datum
    void remove(Date d) {
      // TODO
    }
    /* Liefert das Element an Position index
       requires 0 <= index < Anzahl der Eintraege
     */
    Entry get(int index) {
      // TODO
    }
    // Darstellung als String
    public String toString() {
      // TODO
    }
}
```

Wir zeigen jetzt die Implementierung der einzelnen Methoden im Detail.

Hinzufügen von neuen Einträgen

```
void add(Entry e) {
  Node newNode = new Node(e,null);
  if (first == null) {
    first = newNode;
  } else {
    Node n = first;
    while (n.next != null) {
      n = n.next;
    }
    n.next = newNode;
  }
}
```

- Erstelle zunächst einen neuen Knoten (ohne Nachfolger!).
- Fallunterscheidung
  - List bisher leer: Füge das Element als erstes Element ein.
  - Sonst: Iteriere zum Ende der Liste und füge das Element dort an.
- *Optimierung*: `RunnersDiary`-Objekte könnten zusätzliche eine Referenz auf das letzte Element der Liste verwalten. Dies vereinfacht das Einfügen am Ende der Liste (→ Übungen)

Darstellung als String

```
public String toString() {
  String result = "Lauftagebuch von " + name + "\n";
  Node n = first;
  while (n != null) {
```

```
    result += n.entry.toString() + "\n";
    n = n.next;
  }
  return result;
}
```

- Iteriere, ausgehend vom ersten Knoten, über die Liste und ermittle die String-Repräsentation der Einträge des jeweiligen Knotens.
- Hinweis: Um Strings zusammenzubauen, verwendet man in der Regel `StringBuilder` (siehe Übungen).

Element an bestimmter Position

```
/* Liefert das Element an Position index
   requires 0 <= index < Anzahl der Eintraege
 */
Entry get(int index) {
  Node n = first;
  for (int i = 0; i < index; i++) {
    n = n.next;
  }
  return n.entry;
}
```

- Die Indizes sind, analog zu Arrays, ab 0 nummeriert.
- Zunächst iterieren wir über die Listeneinträge bis zum gewünschten Knoten. Danach wird der Eintrag, der in diesem Knoten abgelegt ist, zurückgeliefert.

Wir können nun Beispiel-Objekte für Lauftagebücher erstellen und mit diesen arbeiten:

```
RunnersDiary diary = new RunnersDiary("Hugo");

diary.add(new Entry(new Date(2,3,2015),5,28,"frisch"));
diary.add(new Entry(new Date(5,7,2015),8.2,40,"k.o."));
diary.add(new Entry(new Date(8,9,2015),10.4,54,"muede"));

StdOut.println(diary);

StdOut.println(diary.get(0));
StdOut.println(diary.get(2));
```