Programmieren in Java

Vorlesung 01: Einfache Klassen

Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2013

Inhalt

Einfache Klassen
- Einfache Klassen am Beispiel
- Aufzählungstypen
- Fallstudie Monopoly
- Methoden für einfache Klassen entwerfen
- Sichtbarkeit
- Main, statische Felder und Methoden, Ein-/Ausgabe

Einführung

Spezifikation

. . . Das Programm soll die Buchhaltung für einen Teegroßhändler unterstützen. Die Quittung für eine Lieferung beinhaltet die Teesorte, den Preis (in Euro pro kg) und das Gewicht der Lieferung (in kg). . . .

Beispielquittungen

- 100kg Darjeeling zu 40.10 EUR
- 150kg Assam zu 27.90 EUR
- 140kg Ceylon zu 27.90 EUR

Modellierung einer Teelieferung

```
1  // Repräsentation einer Rechnung für eine Teelieferung
2  public class Tea {
5      public String kind; // Teesorte
6      public int price; // in Eurocent pro kg
7      public int weight; // in kg
10     public Tea(String kind, int price, int weight) {
11         this.kind = kind;
12         this.price = price;
13         this.weight = weight;
14     }
33 }
```

- Vollständige *Klassendefinition*

Grundgerüst einer Klassendefinition

```
1 // Repräsentation einer Rechnung für eine Teelieferung
2 public class Tea {
```

- Benennt den Klassentyp Tea
- Rumpf der Klasse spezifiziert
  - die Komponenten der *Objekte* vom Klassentyp
  - den *Konstruktor* des Klassentyps
  - das Verhalten der Objekten (später)

```
33 }
```

Felddeklarationen

```
5     public String kind; // Teesorte
6     public int price; // in Eurocent pro kg
7     public int weight; // in kg
```

- Beschreibt die Komponenten: *Instanzvariable*, *Felder*, *Attribute*
- Beschreibung eines Felds
  - Typ des Feldes (String, int)
  - Name des Feldes (`kind`, `price`, `weight`)
- Kommentare: // bis Zeilenende

Konstruktordeklaration

```
10      public Tea(String kind, int price, int weight) {
11          this.kind = kind;
12          this.price = price;
13          this.weight = weight;
14      }
```

- Beschreibt den *Konstruktor*: Funktion, die aus den Werten der Komponenten ein neues Objekt initialisiert
- Argumente des Konstruktors entsprechen den Feldern
- Rumpf des Konstruktors enthält Zuweisungen der Form

this.*feldname* = *feldname*

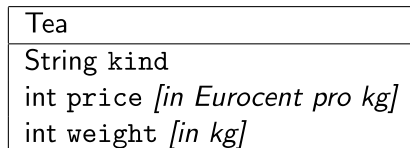
- **this** ist das Objekt, das gerade konstruiert wird
- **this**.*feldname* bezeichnet das entsprechende Feld des Objekts
- *feldname* bezeichnet den Wert des entsprechenden Konstruktorarguments

Beispiel für Teelieferungen

- 100kg Darjeeling zu 40.10 EUR
- 150kg Assam zu 27.90 EUR
- 140kg Ceylon zu 27.90 EUR

```
new Tea("Darjeeling", 4010, 100)
new Tea("Assam", 2790, 150)
new Tea("Ceylon", 2790, 140)
```

Klassendiagramm



- Die Spezifikation einer Klasse kann auch als *Klassendiagramm* angegeben werden.
- Obere Abteilung: Name der Klasse
- Untere Abteilung: Felddeklarationen
- Anmerkung: Klassendiagramme werden in der Softwaretechnik verwendet. Sie sind im UML (Unified Modeling Language) Standard definiert. Sie sind nützliche Werkzeuge für die Datenmodellierung.

Zusammenfassung

- Eine Klasse spezifiziert einen zusammengesetzten Datentyp, den *Klassentyp*.
- Die zum Klassentyp gehörigen Werte sind die *Instanzen* bzw. *Objekte* der Klasse.
- Ein Objekt enthält die Werte der Komponenten in den *Instanzvariablen*.
- Werte vom Klassentyp C werden durch den Konstruktoraufruf

  $$\textbf{new}\ \mathsf{C}(v_1, \ldots, v_n)$$

  gebildet, wobei $v_1, \ldots, v_n$ die Werte der Instanzvariablen sind.

Erstellen einer Klasse

1. Studiere die Problembeschreibung. Identifiziere die darin beschriebenen Objekte und ihre Attribute und schreibe sie in Form eines Klassendiagramms.
2. Übersetze das Klassendiagramm in eine Klassendefinition. Füge einen Kommentar hinzu, der den Zweck der Klasse erklärt. (Mechanisch, außer für Felder mit fest vorgegebenen Werten)
3. Repräsentiere einige Beispiele durch Objekte. Erstelle Objekte und stelle fest, ob sie Beispielobjekten entsprechen. Notiere auftretende Probleme als Kommentare in der Klassendefinition.

Aufzählungstypen

- Ein *Aufzählungstyp* (*enum*) ist definiert durch die Liste seiner Elemente. Modellierung von endlich vielen Zuständen.
- Beispiel

```
1 public enum Day {
2     SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
3     THURSDAY, FRIDAY, SATURDAY
4 }
```

- Verwendungsbeispiel (später mehr)

```
1 public class DayTest {
2     Day weekday;
3     public void reset() {
4         this.weekday = Day.SUNDAY;
5     }
6     public boolean isLectureDay() {
7         return Day.TUESDAY.equals(this.weekday);
8     }
9 }
```

Fallstudie Monopoly

- Modellierung von (Teilen) des Spiels Monopoly
- Sammlung: Bestandteile des Spiels
- Material
  - Regeln: `http://en.wikibooks.org/wiki/Monopoly/Official_Rules`
  - Offizielle Regeln: `http://richard_wilding.tripod.com/monorules.htm`
  - Strassenkarte: `http://monopoly.wikia.com/wiki/Baltic_Avenue`
  - Wikipedia: `http://en.wikipedia.org/wiki/Monopoly_(game)`

Methoden für einfache Klassen entwerfen

Objekte erhalten ihre Funktionalität durch *Methoden*

Beispiel

> *Zu einer Teelieferung (bestehend aus Teesorte, Kilopreis und Gewicht) soll der Gesamtpreis bestimmt werden.*

- Implementierung durch Methode `cost()`
- Keine Parameter, da alle Information im Tea-Objekt vorhanden ist.
- Ergebnis ist ein Preis, repräsentiert durch den Typ int
- Verwendungsbeispiel:
  ```
  Tea tAssam = new Tea("Assam", 2790, 150);
  tAssam.cost()
  ```
  soll 418500 liefern

Methodendefinition

```
// Repräsentation einer Rechnung für eine Teelieferung
public class Tea {
    public String kind; // Teesorte
    public int price; // in Eurocent pro kg
    public int weight; // in kg

    // Konstruktor (wie vorher)
    public Tea (String kind, int price, int weight) { ... }

    // berechne den Gesamtpreis dieser Lieferung
    int cost() { ... }

}
```

- Methodendefinitionen nach Konstruktor
- Methode `cost()`
  - Ergebnistyp int
  - keine Parameter
  - Rumpf muss jetzt ausgefüllt werden

Klassendiagramm mit Methoden

Gleiche Information im Klassenkasten

Tea
`kind` : String `price` : int `weight` : int
`cost()` : int

- Dritte Abteilung enthält die Kopfzeilen der Methoden
  Signaturen von Methoden

Entwicklung der Methode `cost`

- Jede Methode kann auf ihr zugehöriges Objekt über die Variable `this` zugreifen

```
// berechne den Gesamtpreis dieser Lieferung
int cost() { ... this ... }
```

Entwicklung der Methode `cost`

- Jede Methode kann auf ihr zugehöriges Objekt über die Variable `this` zugreifen

```
// berechne den Gesamtpreis dieser Lieferung
int cost() { ... this ... }
```

- Zugriff auf die Felder des Objekts erfolgt mittels `this.`*`feldname`*

```
// berechne den Gesamtpreis dieser Lieferung
int cost() { ... this.kind ... this.price ... this.weight ... }
```

(`kind` spielt hier keine Rolle)

Entwicklung der Methode cost

- Jede Methode kann auf ihr zugehöriges Objekt über die Variable `this` zugreifen

```
// berechne den Gesamtpreis dieser Lieferung
int cost() { ... this ... }
```

- Zugriff auf die Felder des Objekts erfolgt mittels `this.`*`feldname`*

```
// berechne den Gesamtpreis dieser Lieferung
int cost() { ... this.kind ... this.price ... this.weight ... }
```

(`kind` spielt hier keine Rolle)

- Der Rückgabewert der Methode wird durch die **return**-Anweisung spezifiziert.

```
17  // berechne den Gesamtpreis dieser Lieferung
18  public int cost() {
19      return this.price * this.weight;
20  }
```

Methodentest

```
import static org.junit.Assert.*;
import org.junit.Test;
// test cases for Tea
public class TeaTests {
  @Test
  public void testCost() {
    assertEquals(150*1590, new Tea("Assam", 1590, 150).cost());
    assertEquals(220*2790, new Tea("Darjeeling", 2790, 220).cost());
    assertEquals(130*1590, new Tea("Ceylon", 1590, 130).cost());
  }
}
```

- Separate Testklasse, erzeugt mit *New* → *JUnit Test Case*
- Die `import` Statements integrieren den Testrahmen
- Testmethoden werden mit `@Test` annotiert
- Die `assert` Funktion vergleicht den erwarteten Wert eines Ausdrucks mit dem tatsächlichen Wert.

Methoden mit Argumenten

Primitive Datentypen

Der Teelieferant sucht nach billigen Angeboten, bei denen der Kilopreis kleiner als eine vorgegebene Schranke ist.

- Argumente von Methoden werden wie Felder deklariert

```
// liegt der Kilopreis dieser Lieferung unter limit?
boolean cheaperThan(int limit) { ... this ... }
```

- Gewünschtes Verhalten:

```
@Test
public void testCheaperThan() {
  assertFalse(new Tea ("Earl Grey", 3945, 75).cheaperThan (2000));
  assertTrue(new Tea ("Ceylon", 1590, 400).cheaperThan (2000));
}
```

Methoden mit Argumenten

Primitive Datentypen/2

- Methodensignatur

```
// liegt der Kilopreis dieser Lieferung unter limit?
boolean cheaperThan(int limit) { ... this ... }
```

Methoden mit Argumenten

Primitive Datentypen/2

- Methodensignatur

```
// liegt der Kilopreis dieser Lieferung unter limit?
boolean cheaperThan(int limit) { ... this ... }
```

- Im Rumpf der Methode dürfen die Felder des Objekts und die Parameter verwendet werden.

```
// liegt der Kilopreis dieser Lieferung unter limit?
boolean cheaperThan(int limit) { ... this.price ... limit ... }
```

(`kind` und `weight` spielen hier keine Rolle)

Methoden mit Argumenten

Primitive Datentypen/2

- Methodensignatur

```
// liegt der Kilopreis dieser Lieferung unter limit?
boolean cheaperThan(int limit) { ... this ... }
```

- Im Rumpf der Methode dürfen die Felder des Objekts und die Parameter verwendet werden.

```
// liegt der Kilopreis dieser Lieferung unter limit?
boolean cheaperThan(int limit) { ... this.price ... limit ... }
```

(`kind` und `weight` spielen hier keine Rolle)

- Der Rückgabewert der Methode wird durch die **return**-Anweisung spezifiziert.

```
23    public boolean cheaperThan(int limit) {
24        return this.price < limit;
25    }
```

Rezept für den Methodenentwurf

Ausgehend von einer Klasse

1. erkläre kurz den Zweck der Methode (Kommentar)
2. definiere die Methodensignatur
3. gib Beispiele für die Verwendung der Methode
4. definiere die Beispiele als Tests
5. fülle den Rumpf der Methode gemäß dem Muster
   - **this** und die Felder **this**.*feldname* dürfen vorkommen
   - alle Parameter dürfen vorkommen
6. schreibe den Rumpf der Methode

Sichtbarkeit

- **public** : sichtbar im gesamten Programm
- **private** : sichtbar nur in definierender Klasse

Sichtbarkeit

- **public** : sichtbar im gesamten Programm
- **private** : sichtbar nur in definierender Klasse

Sichtbarkeit von Instanzvariablen

- *Datenkapselung*
- generell **private**
- allgemeiner Zugriff über *getter* und *setter* Methoden

Sichtbarkeit

- **public** : sichtbar im gesamten Programm
- **private** : sichtbar nur in definierender Klasse

Sichtbarkeit von Instanzvariablen

- *Datenkapselung*
- generell **private**
- allgemeiner Zugriff über *getter* und *setter* Methoden

Kontrolle über Objekterzeugung

- nur bestimmte Objekte einer Klasse zulässig
- definiere **private** Konstruktor
- Objekterzeugung über statische Methode(n)

Beispiel (Getter und Setter)

```
1  // Repräsentation einer Rechnung für eine Teelieferung
2  public class Tea {
3      private String kind; // Teesorte
4      private int price; // in Eurocent pro kg
5      private int weight; // in kg
6
7      public String getKind() {
8          return this.kind;
9      }
10     public String getPrice() {
11         return this.price;
12     }
13     public String getWeight() {
14         return this.weight;
15     }
16
17     public Tea(String kind, int price, int weight) {
18         this.kind = kind; this.price = price; this.weight = weight;
19     }
20 }
```

Beispiel (Privater Konstruktor)

```
1 public class Tea {
2   ...
3   private Tea(...) { ... } // wie gehabt
4
5   final private static String EARL_GREY = "Earl Grey";
6   public static Tea createEarlGrey (int price, int weight) {
7     return new Tea (EARL_GREY, price, weight);
8   }
9 }
```

Main

Statische Felder und Methoden

- Neben den normalen Feldern und Methoden kann eine Klasse *statische Felder* und *statische Methoden* besitzen.
  (In anderen Sprachen: *Klassenfelder* bzw. *Klassenmethoden*)
- Beide sind *unabhängig* von Objekten und können gelesen, geschrieben und aufgerufen werden, ohne dass ein Objekt der Klasse beteiligt ist.
- Der Zugriff erfolgt mit

```
Klasse.feldname // statisches Feld von Klasse
Klasse.methode (arg...) // statische Methode von Klasse
```

- Beispiel: Die Javabibliothek definiert eine Klasse `Math`, die spezielle Konstanten (e und π) als statische Felder zur Verfügung stellt und trigonometrische und andere Funktionen als statische Methoden bereithält.

```
Math.E, Math.PI
Math.min(4, 5), Math.max(−1, 1), Math.sin(Math.PI / 4)
```

Statische Felder und Methoden

Beispiel

Statische Felder können für Buchhaltungsaufgaben über alle Objekte einer Klasse verwendet werden.

Eine Klasse soll mitzählen, wie oft ihr Konstruktor aufgerufen worden ist.

```
class CountedStuff {
  private static int count = 0;
  private static int inc() {
    return count++;
  }
  private int serial;
  public CountedStuff () {
    this.serial = CountedStuff.inc ();
  }
  public int getSerial () {
    return this.serial;
  }
}
```

Statische Felder und Methoden

Beispiel (Fortsetzung)

```
> CountedStuff x1 = new CountedStuff();
> x1.getSerial()
0
> CountedStuff x2 = new CountedStuff();
> x2.getSerial()
1
> CountedStuff x3 = new CountedStuff();
> x3.getSerial()
2
```

Das Hauptprogramm

- Das Hauptprogramm kann in einer beliebigen Klasse definiert werden.
- Ausgeführt wird die statische Methode `main` mit einem String Array als Parameter, aber ohne Rückgabewert (void).

```
public static void main (String [] arg) {
  ...
}
```

- Das String-Arrays enthält dabei die *Parameter des Programmaufrufs* (z.B. auf der Kommandozeile). Der Aufruf

  `java MyClass eins zwei drei`

  bewirkt, dass im Rumpf von `main`

```
arg.length == 3
arg[0].equals ("eins")
arg[1].equals ("zwei")
arg[2].equals ("drei")
```

Einfache Ausgabe

- Das Objekt `System.out` stellt Methoden zur Ausgabe auf die Konsole bereit.
- Es besitzt (überladene) Methoden `print` für alle primitiven Typen, die jeweils ihr Argument ausdrucken.

```
void print(boolean b)
void print(double d)
void print(int i)
void print(String s)
```

- Die gleichermaßen überladenen Methoden `println` drucken ihr Argument gefolgt von einem Zeilenvorschub.

```
void println() // nur Zeilenvorschub
void println(boolean x)
void println(double x)
void println(int x)
void println(String x)
```

Beispiel: main mit Ausgabe

```
class HelloWorld {
  public static void main (String[] args) {
    System.out.print ("Hello world,");
    for (int i = 0; i < args.length; i++) {
      System.out.print (" " + args[i]);
    }
    System.out.println ();
  }
}
```

Druckt `Hello world,` gefolgt von allen Kommandozeilenargumenten und abgeschlossen mit einem Zeilenvorschub.