Einführung in die Programmierung mit Java

Teil 7: Objektorientiertes Design

Martin Hofmann   Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

28. November 2017

Teil 7: Objektorientiertes Design

1. Objektorientierte Programmierung
   - Fallstudie Game of Life
     - Zerlegung einer Problemstellung in Klassen
     - Entwurfsprinzip Model-View-Controller
   - Packages
   - Spezifikation von Methoden
   - Interfaces (Schnittstellen)

Objektorientiertes Programmierung

IDEE Aufteilung eines komplexen Systems in möglichst eigenständige **Objekte**.

Objekt vereinigt zusammengehörige Daten & Methoden:

Daten oft auch **Felder** oder **Attribute** genannt, in Java durch Instanzvariablen der Klasse dargestellt

Methoden beschreiben Fähigkeiten des Objekts

In Java fassen wir ähnliche Objekte zu Klassen zusammen, im Allgemeinen muss das jedoch nicht der Fall sein.

Wichtig ist größtmögliche **Kapselung**: Jedes Objekt modelliert eine Sache und kümmert sich nur um eigene Daten. Je kleiner die Schnittstelle zu anderen Objekten, desto besser.
In Java erreichen wir dies mit dem Qualifikator `private`

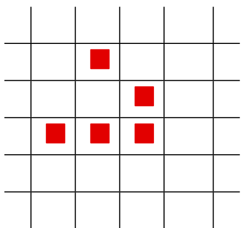
Fallstudie: Game of Life

- Erfunden 1970 von John Horton Conway.
- Spielplan: unendliches 2D-Gitter (für uns: $\infty = 100$ o.ä.)
- Eine Zelle ist entweder lebendig oder tot.
- Zu Beginn sind manche Zellen lebendig andere nicht.
- In jeder Runde ändern sich die Zustände wie folgt:
  - Hat eine lebendige Zelle nur einen oder gar keine lebendigen Nachbarn, so "stirbt" sie in der nächsten Runde.
  - Hat eine lebendige Zelle vier oder mehr lebendige Nachbarn so "stirbt" sie ebenso in der nächsten Runde.
  - Hat eine tote Zelle genau drei lebendige Nachbarn, so wird sie in der nächsten Runde lebendig.
  - Ansonsten ändert sich der Zustand der Zelle nicht.

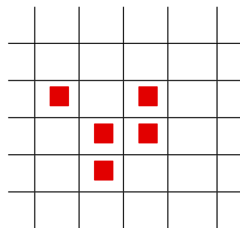
Aufgabe: Simulation des Spiels

Benutzerschnittstelle

- Das Spielfeld wird als $m \times n$ Gitter im Grafikfenster dargestellt.
- Lebendige Zellen werden rot ausgefüllt, tote Zellen weiß.
- Zu Beginn werden die lebendigen Zellen von dem Benutzer mit der Maus angewählt.
- Es werden 1000 Runden simuliert; der zeitliche Abstand der Runden kann eingestellt werden.

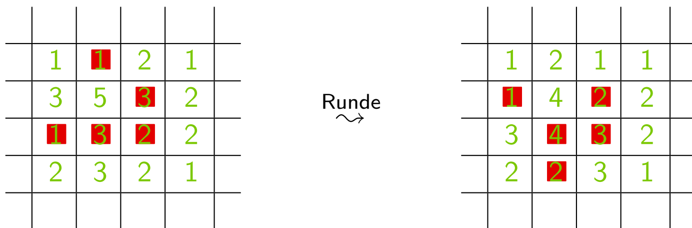


Runde $\rightsquigarrow$



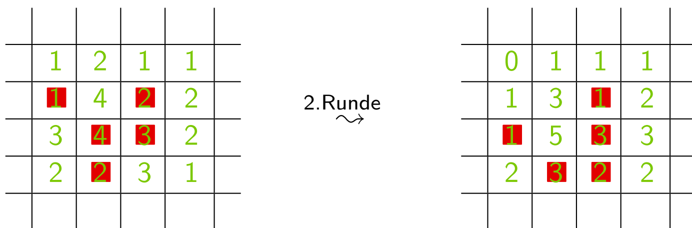
Benutzerschnittstelle

- Das Spielfeld wird als $m \times n$ Gitter im Grafikfenster dargestellt.
- Lebendige Zellen werden rot ausgefüllt, tote Zellen weiß.
- Zu Beginn werden die lebendigen Zellen von dem Benutzer mit der Maus angewählt.
- Es werden 1000 Runden simuliert; der zeitliche Abstand der Runden kann eingestellt werden.



Benutzerschnittstelle

- Das Spielfeld wird als $m \times n$ Gitter im Grafikfenster dargestellt.
- Lebendige Zellen werden rot ausgefüllt, tote Zellen weiß.
- Zu Beginn werden die lebendigen Zellen von dem Benutzer mit der Maus angewählt.
- Es werden 1000 Runden simuliert; der zeitliche Abstand der Runden kann eingestellt werden.



Grundlegende Designprinzipien

- Mögliche Kandidaten für Klassen sind die Substantive der Problembeschreibung.
- Mögliche Kandidaten für Methoden sind die Verben der Problembeschreibung.
- Klassen sollen nicht zu eng gekoppelt sein.
  d.h. Klassen sollten sich nicht gegenseitig benutzen.
- Klassen sollen nicht zu groß werden.
  Meist nicht mehr als 1–3 Bildschirmseiten.
- Großzügig Klassen einführen, keine falsche Sparsamkeit.
- Auf bekannte Entwurfsmuster zurückgreifen.
- Design immer wieder verbessern;
  Architektur mutig umkrempeln code smells → refactoring

Architektur der Fallstudie

- Eine Klasse, die die Parameter (Geometrie, Schnelligkeit, Anzahl der Runden) definiert. Nur statische Instanzvariablen.
- Eine Klasse für Positionen (nicht veränderbar, *immutable*):
  - *Instanzvariablen:* x, y.
  - *Methoden:* x, y abrufen.
- Eine Klasse für Zellen (nicht veränderbar, *immutable*):
  - *Instanzvariablen:* Position, Zustand (tot/lebendig)
  - *Methoden:* Zustand und Position abfragen.
- Eine Klasse für das Spielfeld:
  - *Instanzvariablen:* 2D Array von Zellen
  - *Methoden:* Zelle an einer Position abrufen; Zelle setzen; Nachbarpositionen berechnen; Runde berechnen.
- Eine Klasse für den Spieler (hier "die Natur"):
  - *Instanzvariablen:* Spielfeld, Ansicht.
  - *Methoden:* Spiel initialisieren; Spiel starten/anhalten
- Eine Klasse für die Ansicht:
  - *Instanzvariablen:* GraphicsWindow
  - *Methoden:* Spielfeld & Zelle zeichnen, Position per Click.

Immutable Objects

Objekte, deren Zustand sich nie ändern kann, werden als unveränderlich (engl. **immutable**) bezeichnet.
Z.B.: Falls alle Instanzvariablen der Klasse `final` sind und wiederum ausschließlich auf immutable Objekte zeigen.

Vorteile

Vereinfacht...

- ...Verständnis und Beweis der Korrektheit
  Klasseninvariante muss nur im Konstruktor geprüft werden
- ...direkte Wiederverwendung der Objekte
  defensives Kopieren unnötig
- ...Verwendung in Containern und als Schlüssel in Tabellen
- ...Nebenläufige und Parallele Ausführung
- ...Garbage Collection

$\Rightarrow$ Vieles davon wird erst durch spätere Kapitel klar werden.

Entwurfsprinzip MVC (Model-View-Controller)

Die Aufgaben sind auf drei Komponenten verteilt:

- **Modell**, hier das Spielfeld und die erlaubten Spielzüge.
  Verantwortlich für die Datenhaltung.
- **View**, hier die Ansicht.
  Verantwortlich für die (grafische) Darstellung der Daten.
- **Controller**, hier der Spieler.
  Verantwortlich für das Verarbeiten von Aktionen. Verändert das Modell entsprechend und veranlasst Darstellung der Änderungen bei der Ansicht.

Dieses Architekturprinzip hat sich sehr lange bewährt;
nicht davon abweichen!

Glaubt man, die spezielle Problemstellung erfordere eine andere Architektur, so liegt oft nur ein Mangel an Erfahrung vor.

Umgang mit Zellen am Rand des Spielfelds

- zur Ermittlung des Folgezustands einer Zelle müssen alle acht Nachbarn (N, NO, O, SO, S, SW, W, NW) besucht werden.
- Zellen am Rand des Spielfelds haben aber nur 3 Nachbarn, noch dazu in jeweils unterschiedlichen Richtungen.
- Vergisst man das, so greift man auf nicht existierende Arrayfächer zu $\Rightarrow$ Programmabsturz.
- Explizite Behandlung aller Randfälle mit Fallunterscheidungen ist lästig.
  Aus der Spieleprogrammierung sind zwei Standardlösungen des Problems bekannt:

Berechnung der Nachbarn

Lösung 1: Spezielle Randfelder

Die Randzellen bleiben immer tot; sie werden auch nicht angezeigt und ihr Folgezustand wird nicht neu berechnet. Sie fungieren nur als fiktive Nachbarn des Randes des sichtbaren Felds.

Lösung 2: Wrap-Around

Der linke Nachbar einer Zelle am linken Rand ist per Definition der rechts gegenüberliegende. Mit anderen Worten werden linker und rechter Rand verklebt, sowie auch der obere und untere.
Ein rechteckiges Spielfeld hat nach dieser Verklebung dann die Form eines **Torus** (entspricht Schwimmreifen oder Donut).

Implementierung der Fallstudie

Wir implementieren die Fallstudie während der Vorlesung.

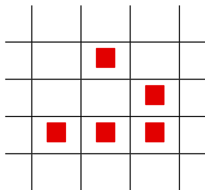
Die endgültige Version wird anschließend auf der Vorlesungsseite bereitgestellt.

Implementierung der Fallstudie

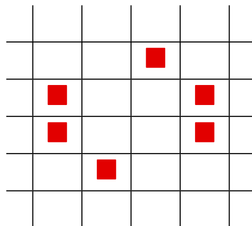
Wir implementieren die Fallstudie während der Vorlesung.

Die endgültige Version wird anschließend auf der Vorlesungsseite bereitgestellt.

Testmuster



Glider



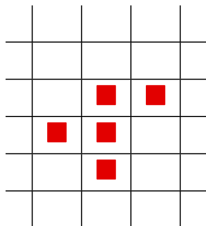
Toad

Implementierung der Fallstudie

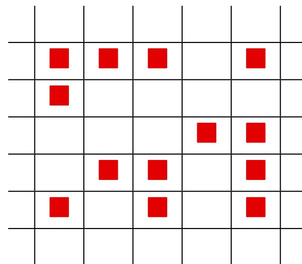
Wir implementieren die Fallstudie während der Vorlesung.

Die endgültige Version wird anschließend auf der Vorlesungsseite bereitgestellt.

Testmuster



R – Pentomino



Block – Engine

Packages

Mehrere Klassen können in Java zu einem Paket (engl. **package**) zusammengefasst werden.

- Jede Datei des Pakets muss mit `package <paketname>;` beginnen, wobei `<paketname>` der **Paketname** ist.
- Die Klassen des Pakets müssen alle in einem Unterverzeichnis liegen, dessen Name dem Paketnamen entspricht.
- Der Paketname sollten nur aus Kleinbuchstaben bestehen.
- Da Paketnamen einzigartig sein sollen, wird empfohlen, die umgekehrte Domain des Herstellers voranzustellen, z.B. `de.lmu.tcs.GraphicsWindow`
- Jeder Punkt im Paktenamen entspricht einem Verzeichnis.

Kompilieren von Packages

Der Java Kompiler erwartet Einhaltung der Verzeichnisstrukturen: Wenn z.B. die Klasse `Main` die Deklaration `package de.lmu.tcs;` enthält, dann muss `Main.java` im Unterverzeichnis `de/lmu/tcs/` gespeichert sein.

Entsprechend im Verzeichnis (Ordner, Directory) darüber kompilieren:

```
./> javac ./de/lmu/tcs/Main.java
./> java de.lmu.tcs.Main
```

. bezeichnet aktuelles Verzeichnis; Windows: / durch \ ersetzen

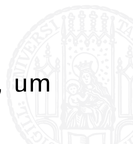
Wie wir hier sehen, muss man sich auch zum Ausführen im Verzeichnis darüber befinden, und die Klasse mit der `main`-Methode mit Paketnamen angeben.

Meistens genügt Kompilieren der Datei mit der `main`-Methode, da benutzte Dateien automatisch mitkompiliert werden.

Import

- Auf die Klassen des Pakets greift man durch Vorschalten von `<paketname>.` zu: Klasse `java.awt.Rectangle` ist völlig unabhängig von Klasse `my.package.Rectangle`
  ⇒ Pakete teilen den Namensraum auf!
- Wenn man anfangs `import <paketname>.*;` schreibt, kann man alle *eindeutigen* Klassennamen auch direkt ohne vorangestellten Paketnamen verwenden.
- Verzeichnisse implizieren dabei *keine* Hierarchie:
  `import java.awt.*;` importiert nicht automatisch auch noch `import java.awt.color.*;`
- Klassen ohne explizite `package`-Deklaration befinden sich in der **default package**. Seit Java 1.4. können solche nicht innerhalb einer echten package verwendet werden.

EXPORT Man kann Pakete in `.jar`-Dateien zusammenpacken, um diese auszutauschen. Direkte Ausführung ebenfalls möglich.



Sichtbarkeit und Pakete

Pakete unterstützen ebenfalls die Aufteilung eines komplexen Systems durch Gruppierung der Einheiten.

- Wird eine gewisse Funktionalität in verschiedenen Softwareprojekten benötigt, so sollte man versuchen, gemeinsame Teile in unabhängige Pakte aufzuteilen.
- Die Klassen eines Paketes sollten daher möglichst untereinander abgeschlossen sein, so dass man jedes Paket einzeln betrachten und verwenden kann.
  $\Rightarrow$ Pakete sollten sich also nicht gegenseitig voraussetzen!

Dies wird durch die Qualifikatoren unterstützt:
Ist eine Klasse, Methode oder Instanzvariable weder `public` noch `private`, so ist sie nur innerhalb ihres Pakets sichtbar.

Zusammenfassung Packages

- Namen des Pakets muss mit Verzeichnisstruktur übereinstimmen
- Pakete dienen zur Aufteilung des Namensraums
- Pakete reduzieren die Schnittstelle durch Einschränkung der Sichtbarkeit:

PRIVATE	Nur innerhalb der Klasse sichtbar
KEIN QUALIFIKATOR	Nur innerhalb des Pakets sichtbar auch als „package-private“ bekannt
PUBLIC	Überall sichtbar

Spezifikation von Methoden: Vor- und Nachbedingungen

- Man kann das Verhalten von Methoden durch eine Vor- und Nachbedingung (wie in der Hoare-Logik) beschreiben.
- Dazu fügt man in der javadoc Dokumentation an geeigneter Stelle eine **Vorbedingung** und eine **Nachbedingung** ein.
- Die Vorbedingung bezieht sich auf die Werte der Parameter der Methode und die Werte der Instanzvariablen vor Ausführung der Methode.

Beispiel

```
public static double sqrt(double a) { ... }
```

Vorbedingung: `a` ist nicht-negativ.
Nachbedingung: Rückgabewert ist $\sqrt{a}$.

Spezifikation von Methoden: Vor- und Nachbedingungen

- Man kann das Verhalten von Methoden durch eine Vor- und Nachbedingung (wie in der Hoare-Logik) beschreiben.
- Dazu fügt man in der javadoc Dokumentation an geeigneter Stelle eine **Vorbedingung** und eine **Nachbedingung** ein.
- Die Vorbedingung bezieht sich auf die Werte der Parameter der Methode und die Werte der Instanzvariablen vor Ausführung der Methode.

Beispiel

```
/**
 * @param a ist nicht negativ
 * @return Quadratwurzel von a
 */
public static double sqrt(double a) { ... }
```

Nachbedingung

- Die Nachbedingung bezieht sich auf die Werte der Instanzvariablen nach Ausführung der Methode und den Rückgabewert, so wie auf die ursprünglichen Parameter vor Ausführung der Methode.
  Man kann auch auf die Werte der Instanzvariablen vor Ausführung der Methode Bezug nehmen durch entsprechende Kennzeichnung, etwa durch `_alt`.
- Vor- und Nachbedingungen sind für uns *informell*.
- Es gibt Werkzeuge wie JML, in denen Vor- und Nachbedingungen formalen Status haben und überprüft werden.

Beispiel

```
/** Abheben.
    @param betrag Der abzuhebende Betrag.
    Vorbedingung:  kontostand >= betrag.
    Nachbedingung: kontostand = kontostand_alt - betrag
    @return Ob Abheben erfolgreich war
*/
public boolean abheben(double betrag) {
    if (kontostand >= betrag){
      kontostand = kontostand - betrag;
      return true;
    else return false;
}
```

Hinweis

Für Vor- und Nachbedingungen gibt es in javadoc leider keine speziellen Tags   im Gegensatz zu `@param`, `@return`, etc.

Es gibt aber erweiterte Tools, welche dies anbieten.

Spezifikation von Methoden und Klassen: Invarianten

Oft soll eine bestimmte Bedingung an die Instanzvariablen von *jeder* Methode der Klasse erhalten werden.
Statt diese in jede Vor- und Nachbedingung aufzunehmen kann man solch eine Bedingung als **Invariante der Klasse** spezifizieren.
Jede Methode muss dann diese Invariante nach ihrer Ausführung garantieren; im Gegenzug darf die Invariante vor Ausführung jeder Methode angenommen werden.

```
/** Klasse für Bankkonten mit Ueberziehungsmoeglichkeit.
    (Invariante: kontostand >= -limit).
*/
public class Bankkonto {
    /** Der Kontostand. */
    private double kontostand;
    /** Ueberziehungslimit. */
    private double limit;
```

Implizite Klasseninvarianten

```
public class Rechteck {
  /**  KLASSENINVARIANTE:
  *    p1 ist linkere obere Ecke, p2 rechte untere Ecke
  */
  private int p1x;
  private int p1y;
  private int p2x;
  private int p2y;
}
```

Geschickte Wahl der Zustandsvariablen kann Klasseninvarianten auch implizit erzwingen (nicht immer möglich):

```
public class Rectangle {
  public int x; // X coordinate of upper-left corner
  public int y; // Y coordinate of the upper-left corner
  public int width;
  public int height;
}
```

Spezifikation von Klassen und Methoden: Zusammenfassung

- Beschreibung des Verhaltens von Methoden und Klassen kann man in Vor- und Nachbedingungen, sowie Invarianten gliedern.
- Die Vorbedingung einer Methode legt Bedingungen an ihren Aufrufkontext fest. Die Methode ist immer so aufzurufen, dass die Vorbedingung erfüllt ist.
- Die Nachbedingung einer Methode spezifiziert den Zustand des Objekts nach Aufruf der Methode. Die Methode ist so zu implementieren, dass die Nachbedingung immer erfüllt ist, vorausgesetzt die Vorbedingung war erfüllt.

- Die Invariante einer Klasse ist eine implizite Vor- und Nachbedingung für alle Methoden einer Klasse: Alle Methoden (einschließlich der Konstruktoren) sind so zu implementieren, dass die Invariante erhalten wird.
- Umgekehrt kann man dann beim Implementieren einer Methode voraussetzen, dass alle Objekte der Klasse die Invariante erfüllen.
- Vor- und Nachbedingungen sind für uns informell. Will man sie formalisieren, so treten nichttriviale Schwierigkeiten auf (exakter Geltungsbereich von Invarianten, Formulierung von Bedingungen ohne Bezugnahme auf private Instanzvariablen, . . .).
- Es existieren solche Formalisierungen, z.B.: JML, die zudem das Erfülltsein von Bedingungen teilweise automatisch überprüfen.

Motivation: Interfaces

Verschiedene Objekte können gleiche Fähigkeiten besitzen, z.B. zwei Objekte der gleichen Klasse haben gleiche Fähigkeiten.

Es kann aber auch sein, dass die Fähigkeiten zweier Objekte nur teilweise überlappen: z.B. haben `Rectangle`, `Dreieck` und `Polygon` alle eine Methode zur Berechnung Ihre Flächeninhalts.

Verschiedene Klassen mit gemeinsamen Fähigkeiten können wir in Java mit einer Schnittstellen-Definition (engl. **interface**) zu einem *neuen Typen* zusammenfassen.

Wenn wir ein Objekt mit einem Interface-Typen erhalten, so kennen wir zwar nicht seine Klasse, aber wir kennen dennoch Methoden, welche wir darauf anwenden können: die Methoden des Interface.

Schnittstellen

Interface-Definition sind sehr ähnlich zu Klassendefinition:

```
public interface HatFläche {
  public static final double PI = 3.1514; // Konstante

  public double berechneFläche();        // Abstrakte M.

  public default double flächeMalPi() {  // Methode
      double f = this.berechneFläche();
      return f * PI;
} }
```

Unterschiede

- Keine Instanzvariablen; nur Konstanten: `static` und `final`
- Keine Konstruktoren
- Methoden dürfen **abstrakt** bleiben,
  d.h. nur Angabe der Typ-Signatur, kein Methodenrumpf!
- `default` Methoden besitzen einen Rumpf als Standardvorgabe

Schnittstellen

Interface-Definition sind sehr ähnlich zu Klassendefinition:

```
public interface HatFläche {
  public static final double PI = 3.1514; // Konstante

  public double berechneFläche();         // Abstrakte M.

  public default double flächeMalPi() {  // Methode
      double f = this.berechneFläche();
      return f * PI;
} }
```

Seit Java 1.9 sind auch `private` `default` Methoden erlaubt.

Unterschiede

- Keine Instanzvariablen; nur Konstanten: `static` und `final`
- Keine Konstruktoren
- Methoden dürfen **abstrakt** bleiben,
  d.h. nur Angabe der Typ-Signatur, kein Methodenrumpf!
- `default` Methoden besitzen einen Rumpf als Standardvorgabe

Benutzung Interface-Typ

Interface `HatFläche` ist ein Typ, der überall dort verwendet werden kann, wo Typen vorkommen:

Beispiel

Bekommt eine Methode einen Parameter des Typs `HatFläche`, so können wir darauf nur die Methoden `berechneFläche` und `flächeMalPi` anwenden, oder die Konstante `PI` abfragen.

```
public double gesamtFläche(HatFläche[] besitztümer) {
  double result = 0;
  for (HatFläche f : besitztümer)
    result = result + f.berechneFläche();
  return result;
}
```

Wie man am Beispiel sieht, kann man auch Arrays über einem Interface-Typen bilden. Die Elemente des Arrays können verschiedenen Klassen angehören — aber alle Elemente haben die Fähigkeiten von `HatFläche`!

Implementierung

Durch die Klausel `implements NameDerSchnittstelle` in einer Klassendefinition wird angezeigt, dass Objekte dieser Klasse die Schnittstelle implementieren.
Objekte so einer Klasse haben dann automatisch *auch* den Typ `NameDerSchnittstelle`.

- Abstrakte Methoden *müssen* implementiert werden.
- Default Methoden können mit `@Override` überschrieben werden gilt auch für statische Methoden
- Konstanten dürfen überschattet werden.

Implementiert eine Klasse die Methoden einer Schnittstelle, fehlt aber die `implements`-Klausel, so haben Objekte der Klasse *nicht* den Typ der Schnittstelle.

Beispiel

```
public class Dreieck implements HatFläche {
  private Point a; private Point b; private Point c;
  public Dreieck(Point a, Point b, Point c) {...}

  public double berechneFläche() {     // Implementiert
    double edgeab = a.distance(b);     // abstrakte Methode
    double edgebc = b.distance(c);
    double edgeca = c.distance(a);
    double umfang = (edgeab + edgebc + edgeca)/2;
    double heron  = Math.sqrt(umfang *(umfang-edgeab)
                        *(umfang-edgebc)*(umfang-edgeca));
    return  heron;
    }

  @Override
  public double flächeMalPi() {
    return PI * this.berechneFläche()
  }
}
```

Beispiel

```
public class Dreieck implements HatFläche {
  private Point a; private Point b; private Point c;
  public Dreieck(Point a, Point b, Point c) {...}

  public double berechneFläche() {      // Implementiert
    double edgeab = a.distance(b);      // abstrakte Methode
    double edgebc = b.distance(c);
    double edgeca = c.distance(a);
    double umfang = (edgeab + edgebc + edgeca)/2;
    double heron  = Math.sqrt(umfang *(umfang-edgeab)
                      *(umfang-edgebc)*(umfang-edgeca));
    return  heron;
    }

  @Override
  public double flächeMalPi() {
    return PI * this.berechneFläche()
  }
}
```

Demonstration @Override-Annotation. Sinnlos ohne Änderung des Methodenrumpfes.

Mehrfache Implementierung von Schnittstellen

Ein Klasse darf auch gleich mehrere Schnittstellen implementieren, wenn sie die Fähigkeiten dieser Schnittstellen jeweils bietet, z.B. `HatFläche` und `HatGewicht`.

Beispiel:

```
class Ball implements HatFläche, HatGewicht, HatFarbe
```

Klasse `Ball` bietet also alle Fähigkeiten (Methoden) der Interfaces `HatFläche`, `HatGewicht` und `HatFarbe` an.

Achtung:

Falls es in diesen Interfaces zufälligerweise eine Methode mit gleichem Namen und gleicher Typsignatur gibt, so *muss* die Klasse `Echt` diese Methode mit `@Override` überschreiben!

Ausblick: Functional Interfaces

Das Paket `java.util.function` stellt viele **Functional Interfaces** bereit. Das sind Interfaces, welche nur eine einzige Methode spezifizieren.

Beispiel

```
interface Comparable<T> {
  int compareTo(T o);
}
```

Idee

Wenn wir ein Array sortieren wollen, so ist der Sortier-Algorithmus unabhängig davon, was wir sortieren — so lange wir die Elemente nur miteinander vergleichen können.
Dieses Functional Interface spezifiziert die Menge aller Klassen, für deren Objekte wir eine Methode zum Vergleichen haben.
Auf dieses wichtige Thema werden wir später noch einmal genauer eingehen, und dann auch erklären, was `<T>` bedeutet.

Sprites

Anderes Beispiel:

```
public interface Sprite {
    /** Zeichnen des Sprite in ein gegebenes
        Rechteck */
    public void zeichnen(GraphicsWindow g, Rectangle r);
}
```

Die Schnittstelle `Sprite` fasst Objekte zusammen, die “sich” in ein gegebenes Rechteck zeichnen können.
Engl.: **sprite** = kleiner Waldgeist.
Bezeichnet kleine Figuren, die sich im Rahmen eines Computerspiels oder einer Animation auf dem Bildschirm umherbewegen.