FAQ + Zusammenfassungen aus der Diskussions-Newsgroup zum Kurs 1618

(zuletzt geändert: 28.05.2011)

Fehlermeldungen

Fehlermeldungen beim Compilieren

1. Missing return statement (in Eclipse: „This method must return a result of type…“)

   Diese Fehlermeldung tritt auf, wenn eine Methode deklariert, einen bestimmten Rückgabetyp zu haben (also nicht als void deklariert ist), der Compiler aber nicht sicherstellen kann, dass tatsächlich ein entsprechender Wert zurückgegeben wird.

   Im einfachsten Fall liegt das daran, dass schlicht eine entsprechende Return-Anweisung fehlt.

   Es kann aber auch sein, dass die Return-Anweisung zwar vorhanden ist, die Methode aber auch regulär beendet werden kann, ohne dass die Anweisung ausgeführt wird. Ein typisches Beispiel ist das folgende, in dem im Fall einer Division durch 0 die auftretende Exception abgefangen wird – die Methode also nicht abrupt beendet wird, sondern regulär – aber im Catch-Block eben kein Wert zurückgegeben wird:

   ```
   int dividiere(int dividend, int divisor) {
      try {
         return dividend / divisor;
      } catch (ArithmeticException e) {
         e.printStackTrace();
      }
   }
   ```

2. Non-static method / field … cannot be referenced from a static context (in Eclipse: „Cannot make a static reference to the non-static method / field…“)

   Siehe dazu im Kapitel „Variable / Attribut / Klassenattribut / lokale Variable“ den Abschnitt „Warum darf eine Klassenmethode nur auf Klassenattribute zugreifen?„ (das dort bzgl. Attributen Gesagte gilt analog für Exemplarmethoden) sowie das komplette Kapitel „Verwendung von static„.

3. „The method m() of type X should be tagged with @Override since it actually overrides a superinterface / superclass method“ (Warnung in Eclipse)

   Seit Java 1.6 gibt es in Java die Möglichkeit, sog Annotations zu verwenden. Von diesen sind einige bereits „in Java eingebaut“, eine davon ist @Override. Wenn Sie diese

Annotation vor eine Methodendeklaration setzen, dokumentieren Sie damit explizit, dass Ihre Methode eine Methode eines Supertyps überschreibt. Sollte das dann aber gar nicht der Fall sein, etwa, weil Sie versehentlich eine andere Parameteranzahl oder andere Parametertypen gewählt haben als bei der Methode, die Sie überschreiben wollten, wird der Compiler Ihnen eine Fehlermeldung geben.

Um nun das volle Potential von @Override auszunutzen, ist es sinnvoll, diese Annotation konsequent zu verwenden und sich ein *fehlendes* @Override bei einer Methode, welche eine Superklassenmethode überschreibt, zumindest als Warnung melden zu lassen. Bei dem Eclipse-Workspace, den Sie auf http://feu.mpaap.de/eclipse/index.html herunterladen können, ist diese Einstellung bereits vorgenommen und wenn Sie in einem Subtyp einer Methode eines Supertyps überschreiben, ohne diese Absicht durch @Override explizit zu dokumentieren, erhalten Sie die oben genannte Warnung.

Fehlermeldungen beim Ausführen

1. NullPointerException (NPE)

   Eine NPE tritt genau dann auf, wenn man versucht, das „Objekt hinter einer Referenz“ anzusprechen, also zu „dereferenzieren“, man aber in Wirklichkeit die leere Referenz „null“ vor sich hat, es also gar kein referenziertes Objekt gibt.

   Dieses Dereferenzieren findet statt, wenn auf eine Methode oder ein Attribut eines Objekts zugegriffen werden soll. Man kann den Punkt als Dereferenzierungsoperator lesen, analog den entsprechenden Operatoren z.B. in C oder Pascal. Außerdem findet eine Dereferenzierung statt, wenn man auf ein Element eines Arrays zugreift: Auch ein Array ist ja ein Objekt und eine Referenz, von der man glaubt, sie verweise auf einen Array, kann natürlich auch den Wert null haben.

   Wenn man eine NullPointerException bekommt, ist der erste Schritt also, herauszufinden, *welche* Referenz, die man zu dereferenzieren versucht, eigentlich den Wert null hat. Dazu bietet es sich an, die betreffende Zeile so zu zerlegen, dass man pro Zeile nur noch eine Dereferenzierung hat. Lautet die Zeile, in der die NPE auftritt beispielsweise

   ```
   String name = allPersons.getPersonArray()[2].getName();
   ```

   gibt es drei Dereferenzierungen und man könnte die Zeile wie folgt zerlegen:

   ```
   Person[] persons = allPersons.getPersonArray();
   Person person = persons[2];
   String name = person.getName();
   ```

   Führt man nun das Programm erneut aus, kann man diesmal an der Zeilennummer der Exception genau erkennen, welche Referenz den Wert null hatte. Dann muss man „nur“ noch herausfinden, warum sie diesen Wert hat.

Ein paar typische Ursachen:

- Man hat ein Attribut eines Referenztyps zwar deklariert, aber nicht initialisiert, ihr also kein Objekt zugewiesen. Damit hat es per automatischer Initialisierung den Wert null.

- Ein Attribut eines Referenztyps sollte an einer anderen als der Deklarationsstelle initialisiert werden, dabei wurde aber versehentlich eine erneute Deklaration vorgenommen, also eine lokale Variable deklariert. Die Initialisierung betrifft dann natürlich diese lokale Variable; das Attribut behält den Wert null.

- Ein Referenztyp-Array wurde zwar erzeugt, die einzelnen Ojekte aber nicht. Damit haben die Elemente des Arrays per automatischer Initialisierung den Wert null. Näheres siehe hier.

- Man will auf ein Attribut zugreifen, hat aber versehentlich eine lokale Variale oder einen formalen Parameter gleichen Namens deklariert. Diese Variable verdeckt das Attribut, so dass man auf die lokale Variable bzw. den formalen Parameter zugreift, die dann natürlich einen anderen Wert hat, als man erwartete, z.B. null.

2. ArrayIndexOutOfBoundsException (AIOOBE)

   Siehe hierzu im Kapitel „Arrays“ den Abschnitt „Ich bekomme eine ArrayIndexOutOfBoundsException. Was hat es damit auf sich?„

3. ClassNotFoundException (CNFE)

   Eine CNFE bedeutet, dass die Laufzeitumgebung eine Klasse nicht laden konnte, welche sie zur Ausführung des Programms benötigte.

   Die häufigste Ursache dieser Exception ist ein falsch gesetzter Classpath. Der Classpath sagt der Laufzeitumgebung, wo sie nach Klassen zu suchen hat. Ist der Classpath falsch gesetzt, werden z.B. Klassen im aktuellen Verzeichnis nicht gefunden, wo normalerweise automatisch gesucht wird.

   Die systemweite Umgebungsvariable CLASSPATH sollte aber aus verschiedenen Gründen ohnehin nicht gesetzt werden. Wenn zu bestimmten Zwecken eine explizite Angabe des Classpath nötig ist, sollte dazu beim Start des Programms der Aufrufparameter -classpath verwendet werden.

   Eine weitere Ursache besteht darin, dass bei der Exemplarerzeugung per Reflection (z.B. im Adressbuchbeispiel aus der Newsgroup) der Methode forName() der Klasse Class ein String als Klassenname übergeben wird, dem die Laufzeitumgebung keine Klasse zuordnen kann. Wenn es sich nicht um einen einfachen Tippfehler handelt, wurde hier meist die Package-Angabe vergessen, die zum vollständigen Klassennamen aber dazugehört.

Im Kontext des Kurses 1618 begegnet einem die CNFE meist im Zusammenhang mit RMI (Kurseinheit 7). Dazu einige Erläuterungen:

Damit die RMI-Registry (welche selbst ein Java-Programm ist) die benötigten Server-Klassen findet, gibt es mehrere Möglichkeiten:

1. Man benötigt die RMI-Registry nur für genau ein Serverprogramm

Dann kann man sie aus dem Verzeichnis heraus starten, aus dem man auch das Serverprogramm startet (unter der Annahme, dass über diesen Pfad auch das Remote-Interface erreichbar ist). Einfacher ist es dann aber, die Registry direkt aus dem Serverprogramm zu starten. Dazu genügt folgende Zeile:

```
java.rmi.registry.LocateRegistry.createRegistry(1099);
```

Nachteil: Diese RMI-Registry wird zusammen mit dem Serverprogramm beendet, deswegen ist diese Lösung ungeeignet, wenn mehrere Programme die Registry nutzen sollen.

2. Die Registry wird für mehrere Serverprogramme verwendet.

In diesem Fall ist das Serverprogramm dafür zuständig, der Registry mitzuteilen, wo die von ihr benötigten Klassen liegen. Dazu übergibt man beim Start des Serverprogramms diese Information mit Hilfe des VM-Parameters -Djava.rmi.server.codebase. Das könnte z.B. so aussehen:

```
java -Djava.rmi.server.codebase=file:/D:/1618/bin/pufferServer.RingPufferServer
```

Wenn es sich bei der übergebenen Codebase um ein Verzeichnis handelt, muss an dessen Ende ein Slash stehen.

In der Praxis wird man oft wesentlich komplexere Szenarien haben, bei denen es erwünscht ist, dass die zum Client und/oder zum Server gehörigen Klassen dynamisch von der jeweils anderen oder von dritter Seite geladen werden können. RMI unterstützt derlei durch die Möglichkeit, Klassen dynamisch von Servern laden zu können. Einen guten Überblick über das dynamische Laden von Klassen im Kontext von RMI bietet diese Seite. Das dort Beschriebene geht allerdings weit über die Anforderungen des Kurses hinaus.

4. IllegalMonitorStateException (IMSE)

Eine IMSE tritt genau dann auf, wenn ein Thread wait(), notify() oder notifyAll() auf einem Objekt aufruft, dessen Monitor dieser Thread nicht besitzt. Der Aufruf der besagten Methoden richtet sich ja immer an ein ganz bestimmtes Objekt, auf dessen Warteschlange er sich bezieht.

Der einfachste solche Fall ist, dass dass man eine der drei Methoden in einem Bereich aufruft, der überhaupt nicht synchronisiert ist.

Dann gibt es den Fall, dass man eine der drei Methoden zwar in einem synchronisierten Bereich aufruft, aber auf einem anderen Objekt als dem (bzw. einem von denen) auf dem (bzw. denen) der Bereich synchronisiert ist.

Von diesem zweiten Fall gibt es noch eine besonders subtile Variante, nämlich dass man einer der Methoden zwar auf der Variablen aufruft, über die auch die Synchronisation erfolgte, diese Variable aber inzwischen ein anderes Objekt referenziert. Um dies zu verhindern, werden zur Synchronisation, so weit sie nicht auf „this“ erfolgt, meist als final deklarierte Variablen (also Konstanten) verwendet, da sich hier das referenzierte Objekt nicht ändern *kann*. Vorsicht in diesem Zusammenhang bei Arrays: Die Deklaration einer Variablen eines Array-Typs verhindert ja *nicht*, dass den Elementen des Arrays neue Werte zugewiesen werden.

Arrays

Grundlagen

Hier zunächst kurz die wichtigsten Fakten, die man über Arrays in Java wissen sollte:

1. Array-Indexe zählen ab 0
2. Arrays sind Objekte. Ihre Elemente verhalten sich in vielerlei Hinsicht wie die Attribute anderer Objekte.
3. Die Elemente eines Arrays, dessen Basistyp ein Referenz-Typ ist, sind Referenzen, nicht Objekte.
4. Mehrdimensionale Arrays in Java sind Arrays, deren Elemente Referenzen auf weitere Arrays sind.

Ich bekomme eine ArrayIndexOutOfBoundsException. Was hat es damit auf sich?

Eine AIOOBE tritt auf, wenn man versucht, über einen ungültigen Index auf ein Element eines Arrays zuzugreifen. In der mit der Exception gelieferten Meldung steht, welchen Wert der Index bei dem problematischen Zugriffsversuch hatte. Da Java-Arrays mit dem Index 0 beginnen ist der größte mögliche Index um eins kleiner als die Länge des Arrays.

Typische Ursache für eine AIOOBE sind falsche Obergrenzen bei For-Schleifen. Eine korrekte For-Schleife über alle Elemente eines Arrays sieht so aus (man beachte das Kleiner-Zeichen in der Schleifenbedingung, im Gegensatz zu dem von Pascal – wo Arrays ab 1 zählen – gewohnten Kleiner-Gleich):

```
for (int i = 0; i < myArray.length; i++) {
   System.out.println(myArray[i]);
}
```

Der eleganteste Weg, eine AIOOBE zu vermeiden, besteht darin, gar keinen expliziten Index zu verwenden, sondern die sog. ForEach-Form der For-Schleife zu benutzen (was leider nicht immer möglich ist):

```
for (Person person : myArray) {
   System.out.println(person);
}
```

Näheres dazu im Abschnitt zum Thema ForEach-Form der For-Schleife

Ich bekomme beim Zugriff auf ein Array-Element eine NullPointerException. Warum?

Die typische Ursache ist, dass Sie zwar den Array selbst erzeugt haben, nicht jedoch seine Elemente. Eine Codezeile wie

```
   Vogel[] alleVoegel = new Vogel[42];
```

erzeugt eben nur den Array, nicht jedoch irgendwelche vordefinierten Vogel-Exemplare. Das mag auf den ersten Blick überraschen, wird aber völlig logisch, wenn man sich klarmacht, dass der Compiler kaum dem Programmierer die Entscheidung abnehmen kann, welche Objekte er eigentlich in den Array packen möchte und wie diese zu initialisieren sind. Im Falle der Vögel ist es z. B. recht wahrscheinlich, dass diese Klasse abstrakt ist, es also überhaupt keine direkten Vogel-Exemplare geben kann.

Übrigens verhalten sich Arrays hier völlig analog zu anderen Objekten: Alle Attribute, die nicht explizit initialisiert werden erhalten bei der Erzeugung des Objekts zunächst den Default-Wert des entsprechenden Typs und bei Referenztypen ist dies nun einmal die leere Referenz null.

Ich verwende eine Kopie eines Arrays, aber wenn ich darin Änderungen vornehme, wirken sich diese auf das Original aus. Warum?

Verwenden Sie vielleicht gar keine Kopie? Evtl. haben Sie die vermeintliche Kopie etwa so erzeugt:

```
Vogel[] kopie = alleVoegel;
```

Damit erzeugen Sie aber keinen neuen Array, sondern Sie deklarieren lediglich eine neue Variable „kopie“ , welche denselben Wert hat wie „alleVoegel“. Der Wert der Variablen ist aber *nicht* der Array, sondern lediglich eine Referenz auf diesen. Diese haben Sie kopiert, d.h. Sie haben also jetzt zwei Variablen, die dasselbe Objekt referenzieren, sogenannte Aliase.

Aber auch wenn ich den Array explizit kopiere, wirken sich Änderungen auf das Original aus!

Nein. Änderungen *am kopierten Array* wirken sich nicht auf das Original aus. Wahrscheinlich entspricht Ihr Problem dem folgenden Beispiel:

```
Person[] original = new Person[2];
original[0] = new Person("Donald", "Duck");
original[1] = new Person("Franz", "Gans");
Person[] kopie = new Person[2];
System.arraycopy(original, 0, kopie, 0, 2);
kopie[1].setVorname("Gustav");
System.out.println(original[1].getVorname());
System.out.println(kopie[1].getVorname());
```

Hier erhalten Sie in der Tat zwei mal die Ausgabe „Gustav“. Die Ursache ist aber *nicht*, dass sich Änderungen an „kopie“ auf „original“ auswirken, sondern, dass die Elemente beider Arrays Referenzen auf dieselben Objekte enthalten, kopie[1] also dasselbe Objekt referenziert wie original[1]. Um sich den Unterschied klarer zu machen, testen Sie einmal folgenden Code, welcher tatsächlich Änderungen *am kopierten Array* vornimmt.

```
Person[] original = new Person[2];
original[0] = new Person("Donald", "Duck");
original[1] = new Person("Franz", "Gans");
Person[] kopie = new Person[2];
System.arraycopy(original, 0, kopie, 0, 2);
kopie[1] = new Person("Gustav", "Gans");
System.out.println(original[1].getVorname());
```

```
System.out.println(kopie[1].getVorname());
```

Die Dienstleistungsmethode System.arraycopy() macht also letztlich nichts anderes als das, was Sie auch mit einer for-Schleife machen könnten: Sie kopiert Elemente um. Und da die Elemente eines Arrays, der „Objekte enthält“ nun einmal gar nicht die Objekte sind, sondern nur Referenzen auf diese, werden auch nur Referenzen kopiert... die dann natürlich dieselben Objekte referenzieren wie ihre Originale.

Kann ich mit Hilfe von System.arraycopy() auch mehrdimensionale Arrays kopieren?

Ja, aber damit erreichen Sie nicht das, was Sie vermutlich wollen. Denn ein mehrdimensionaler Array ist in Java nur ein Array, dessen Elemente Referenzen auf weitere Arrays sind. Und beim Kopieren der „ersten Ebene“ würden nur die Referenzen auf die Arrays der zweiten Ebene kopiert, nicht aber diese selbst, siehe auch die vorigen beiden Fragen dieses Kapitels. Wenn Sie eine „tiefe“ Kopie eines mehrdimensionalen Arrays haben wollen, müssen die die äußeren Ebenen „von Hand“ kopieren, also mit (ggf. geschachtelten) for-Schleifen. Für die innerste Ebene können Sie dann statt einer for-Schleife natürlich System.arraycopy() einsetzen.

ForEach-Form der For-Schleife

Wie funktioniert die ForEach-Form der For-Schleife prinzipiell?

Antwort: Mit dieser Form der Schleife kann man über Arrays und über Subtypen des Interfaces Iterable (typischerweise sind das Collections) iterieren, ohne explizit eine Indexvariable oder einen Iterator anzugeben. Man gibt stattdessen Namen und Typ einer lokalen Variablen an, in die dann automatisch bei jedem Schleifendurchlauf der jeweils nächste Wert des Arrays bzw. des Iterable kopiert wird. Codebeispiel:

```
import java.util.LinkedList;

public class Test {
   public static void main(String[] args) {

      // Array-Initialisierung mit "klassischer" For-Schleife
      String[] array = new String[3];
      for (int i = 0; i < array.length; i++) {
         array[i] = "Wert";
      }

      // Iteration über den Array mit ForEach-Schleife
      for (String element : array) {
         System.out.println(element);
      }

      System.out.println();

      // Initialisierung einer LinkedList mit den Elementen
      // des Arrays mit Hilfe einer ForEach-Schleife
      LinkedList<String> list = new LinkedList<String>();
      for (String element : array) {
         list.add(element);
      }

      // Iteration über die LinkedList mit ForEach-Schleife
      for (String element : list) {
         System.out.println(element);
      }

      System.out.println();

      // Aber Achtung: Eine *Zuweisung* an die lokale
      // Iterationsvariable bleibt wirkungslos!
      for (String element : array) {
         element = "Neuer Wert";
      }

      // Wie man sieht: Die Werte im Array sind unverändert
      for (String element : array) {
         System.out.println(element);
```

```
    }
  }
}
```

Kann ich in einer ForEach-Schleife keine Zuweisung an das aktuelle Element vornehmen?

Nein, das geht nicht. Die Variable, in welche die Elemente der Iteration kopiert werden, verhält sich hier ähnlich dem Parameter eines Methodenaufrufs: Es wird jeweils der Wert des „Originals“ (von außerhalb des Schleifenrumpfs) hineinkopiert (Siehe dazu auch den Abschnitt „Call by Value / Call by Reference“). Eine Zuweisung an die lokale Variable verändert also nicht das Iterable, über das iteriert wird. Wenn der Wert eine Objektreferenz ist, kann aber selbstverständlich über die lokale Kopie der Referenz das referenzierte Objekt (durch Methodenaufruf oder Attributzuweisung) verändert werden.

Warum bekomme ich eine ArrayIndexOutOfBoundsException beim Iterieren über einen Array?

Hier der problematische Quellcode:

```
for (int i : myArray) {
   System.out.println(myArray[i]);
}
```

Antwort: Die ForEach-Form der For-Schleife arbeitet nicht mit sichtbaren Indexzahlen. Die lokale Variable enthält also nicht, wie bei der klassischen Form, die Indexzahlen der Elemente, sondern vielmehr (eins nach dem anderen) die Werte der Elemente selbst. Offenbar ist eines dieser Elemente in deinem Beispiel größer als der größte erlaubte Index des Arrays. Richtig wäre also (ich nenne die temporäre Variable hier bewusst nicht i, sondern elem, um anzudeuten, dass es sich nicht um eine Indexzahl handelt):

```
for (int elem : myArray) {
   System.out.println(elem);
}
```

Call by Value / Call by Reference

Werden Parameter in Java „by value“ oder „by reference“ übergeben?

In den Weiten des Internet finden sich zu diesem Thema die merkwürdigsten Aussagen, u.a. die, in Java würden Primitivtypen „by value“, Objekte hingegen „by reference“ übergeben. Das ist Unsinn: Parameterübergaben erfolgen in Java *immer* „by value“, d.h. es wird eine Kopie des übergebenen Wertes dem formalen Parameter zugewiesen. Diese Aussage bedarf allerdings einer zusätzlichen Erläuterung, schließlich hat die teilweise existierende Verwirrung ja ihren Grund.

Zunächst einmal: Zugriff auf Objekte hat man in Java *ausschließlich* über Referenzen. Variablen enthalten nie Objekte, sondern – Primitivtypen mal außen vor gelassen – immer nur Referenzen, Methoden liefern keine Objekte, sondern Referenzen. Wenn also einer Methode „ein Objekt als Parameter übergeben“ wird, wird in Wirklichkeit eine Referenz übergeben.

Und damit wird die obige Aussage verständlich: Wenn der übergebene Wert eine Referenz ist, wird eben eine *Kopie der Referenz* erzeugt und dem formalen Parameter zugewiesen! Diese Kopie verweist natürlich auf genau das Objekt, auf das auch die Original-Referenz verweist.

Wenn man das verstanden hat, wird auch klar, warum eine *Zuweisung* an den formalen Parameter keine Wirkung nach außen erzeugt: Dieser referenziert danach schlicht ein anderes Objekt. Hingegen können Methodenaufrufe oder schreibende Attributzugriffe über den formalen Parameter sehr wohl den Zustand des referenzierten Objekts verändern! Denn bei solchen Zugriffen erfolgt ja eine Dereferenzierung, man greift also auf das referenzierte Objekt zu! Und damit werden *solche* Veränderungen auch außerhalb der Methode wirksam.

Man sollte sich – gerade wenn man Erfahrung in Sprachen wie C hat – klar machen, dass das eben *nicht* dem Verhalten entspricht, welches man erwarten müsste, wenn es stimmte, dass Objekte „by reference“ übergeben würden!

Und wie ist das bei Arrays?

Arrays sind in Java Objekte. Eine Variable, welche „einen Array enthält“, enthält in Wirklichkeit also lediglich eine Referenz auf ein solches Objekt. Wird an eine Methode „ein Array übergeben“, wird wie oben beschrieben, die Referenz „by value“ übergeben und der methodenlokalen Variablen zugewiesen, wie bei jedem anderen Objekt auch.

Wenn man nun über eine Array-Referenz auf ein Element des Arrays zuzugreift, erfolgt ebenso eine Dereferenzierung, wie beim Zugriff auf eine Methode oder ein Attribut eines „normalen“ Objekts. Man kann sich die Elemente eines Arrays als Attribute des Array-Exemlars vorstellen, nur dass die Selektion nicht über den Punkt (= die Dereferenzierung), gefolgt vom Attributnamen passiert, sondern mit einer speziellen Syntax.

Und so ist auch das Verhalten völlig analog dem anderer Objekte: Erfolgt innerhalb einer Methode eine Zuweisung an ein *Element* eines übergebenen Arrays, wirkt sich diese sehr wohl nach außen aus, denn beim Elementzugriff wird dereferenziert, und der referenzierte Array ist ja innen wie außen derselbe. Eine Zuweisung an die lokale Variable selbst hingegen bleibt natürlich nach außen wirkungslos.

```
public class ArrayTest {
   public static void main(String[] args) {
      String[] array = { "eins", "zwei" };

      test(array);
      for (String elem : array) {
         System.out.println(elem);
      }

      swap(array);
      for (String elem : array) {
         System.out.println(elem);
      }
   }

   static void test(String[] array) {
      array = new String[] { "Alles", "ganz", "anders" };
   }

   static void swap(String[] array) {
      String temp = array[0];
      array[0] = array[1];
      array[1] = temp;
   }
}
```

Die Ausgabe lautet:

```
eins
zwei
zwei
eins
```

Casts

Wie kann ich den Typ X auf Y casten?

Da das Wesen eines Casts oft missverstanden wird, hier eine Erläuterung dazu, was ein Cast eigentlich ist, wobei ich Primitivtypen außen vor lasse.

Einen Cast kann man sich als eine Zusicherung an den Compiler vorstellen, dass ein Ausdruck eines Referenztyps (meist eine Variable) zur Laufzeit ein Objekt eines bestimmten Typs referenzieren wird.

Ein einfaches Beispiel:

```
public class Test {
   public static void main(String[] args) {
      Person[] persons = new Person[4];
      for (int i = 0; i < persons.length; i++) {
         persons[i] = new Student(i);
      }
      for (int i = 0; i < persons.length; i++) {
         System.out.println(persons[i].getMatrikelNr());
      }
   }
}

abstract class Person {
   /* ... */
}

class Professor extends Person {
   /* ... */
}

class Student extends Person {

   private int matrikelNr;

   public Student(int matrikelNr) {
      this.matrikelNr = matrikelNr;
   }

   int getMatrikelNr() {
      return matrikelNr;
   }
}
```

Hier meldet der Compiler einen Fehler bzgl. des Methodenaufrufs

```
persons[i].getMatrikelNr()
```

nämlich: „The method getMatrikelNr() is undefined for the type Person“. Das liegt daran, dass für den Compiler die Elemente des Arrays persons vom Typ Person sind. Und für diesen Typ gibt es tatsächlich keine Methode getMatrikelNr(). Dass der Array zur Laufzeit Student-Exemplare enthalten wird, kann der Compiler nicht wissen. Wir können es ihm aber zusichern, nämlich durch einen Cast. Der Aufruf sähe dann so aus:

```
((Student) persons[i]).getMatrikelNr();
```

Hiermit versichern wir dem Compiler, dass persons[i] zur Laufzeit eine Referenz auf ein Objekt vom Typ Student enthalten wird und der Compiler erlaubt nun die Verwendung der Student-Methode. Eine Änderung des Typs eines Objekts erfolgt dabei nicht und kann auch nicht erfolgen: Ein Objekt behält für sein ganzes Leben seinen Typ, mit dem es per „new“ erzeugt wurde.

Wem die Vorstellung von „Zusicherungen“ an den Compiler nicht gefällt, der kann sich vielleicht mit der folgenden eher formalen Sichtweise anfreunden:

Der Compiler zieht für seine Typprüfungen ausschließlich die Deklarationstypen der beteiligten Ausdrücke heran, Ein Cast nun bildet zusammen mit dem Ausdruck, vor dem er steht, einen neuen Ausdruck, dessen Deklarationstyp genau der Typ ist, auf den gecastet wird. Dazu ein Beispiel:

```
Person p = new Student(123456);
Student s = (Student) p;
```

Hier ist p ein Ausdruck, dessen Deklarationstyp der Typ Person ist. Der Ausdruck (Student) p hingegen hingegen hat definitionsgemäß den Deklarationstyp Student, weswegen der Compiler die Zuweisung an die Variable s zulässt.

Der Compiler lässt mich eine Methode nicht aufrufen, obwohl das Objekt diese hat! Warum nicht?

Beispielcode:

```
public class Test {
   public static void main(String[] args) {
      Super sup = new Sub();
      sup.m();
   }
}

class Super {

}

class Sub extends Super {
```

```
    void m() {
        System.out.println("Hello");
    }
}
```

Antwort: Der Compilerfehler in der Zeile

```
sup.m();
```

entsteht, weil versucht wird, auf einem Ausdruck vom Deklarationstyp Super eine Methode aufzurufen, die für diesen Typ nicht definiert ist („The method m() is undefined for the type Super“). Dass die Variable sup vielleicht zur Laufzeit ein Sub enthalten wird, weiß der Compiler nicht, er prüft lediglich die Zulässigkeit der Zuweisung

```
Super sup = new Sub();
```

anhand der Deklarationtypen links und rechts des Zuweisungsoperators. Mit einem Cast

```
((Sub) sup).m();
```

ändert sich der Deklarationstyp des Ausdrucks, auf dem die Methode aufgerufen wird zu Sub und der Compiler erlaubt den Aufruf. Man kann sich das auch so vorstellen, dass man dem Compiler durch den Cast zusichert, dass sup zur Laufzeit ein Sub-Exemplar referenzieren wird. Eine solche Zusicherung hebelt natürlich letztlich die Typprüfung durch den Compiler aus: Sollte sich zur Laufzeit dann herausstellen, dass sup gar kein Sub referenziert, kommt es zu einer ClassCastException.

Variable / Attribut / Klassenattribut / lokale Variable

Was ist der Unterschied zwischen Variablen und Attributen?

„Variable“ ist eigentlich der allgemeinere Begriff. Auch ein Attribut ist eine Variable. Allerdings wird in der Literatur (leider) oft „Variable“ verwendet wenn methoden- oder blocklokale Variablen (also keine Attribute) gemeint sind. Aus dem Kontext wird das dann mehr oder weniger gut klar. Besser ist es auf jeden Fall, genau zu schreiben, was man meint. Hier eine kleine, mehr oder weniger sinnfreie Beispielklasse zur Erläuterung der Begriffe:

```
public class Person {

   public static int maxPersonalNummer

   private String name;

   private String vorname;

   private int geburtsJahr;

   private final int personalNummer;

   public Person(String vorname, String name, int geburtsJahr) {
      this.name = name;
      this.vorname = vorname;
      this.geburtsJahr = geburtsJahr;
      Person.maxPersonalNummer++;
      personalNummer = Person.maxPersonalNummer;
   }

   public int getAlter() {
      int aktuellesJahr = IrgendeineKlasse.getCurrentYear;
      return aktuellesJahr - geburtsJahr;
   }
}
```

Attribut: Eine Variable, die Teil des Zustands eines Objekts ist. Ein Attribut wird über sein Objekt angesprochen. Aus einem Objekt heraus werden seine Attribute über „this“ angesprochen, das „this“ kann aber auch wegfallen, wenn der Bezeichner nicht durch eine methodenlokale Variable verdeckt wird. Im Beispiel sind name, vorname, geburtsJahr und personalNummer Attribute, wobei personalNummer eine Sonderrolle spielt, weil es nach der Initialisierung unveränderlich ist. Attribute werden in der Klasse für deren Exemplare deklariert, gehören aber den Exemplaren, jedes Exemplar hat sein eigenes.

Klassenattribut: Eine einer Klasse zugeordnete Variable, die nicht zu einem Exemplar der Klasse gehört, sondern zur Klasse selbst. Klassenattribute spielen in Java oft die Rolle, die in der prozeduralen Programmierung globale Variablen spielen. Ein Klassenattribut ist erkennbar am Modifikator „static“ und wird über den Namen der Klasse angesprochen. Bei Zugriffen aus derselben Klasse kann die Angabe des Klassennamens entfallen.

Methodenlokale Variable: Eine Variable, die innerhalb einer Methode deklariert ist. Eine methodenlokale Variable ist ab der Deklarationsstelle innerhalb der Methode bis zu deren Ende gültig. Im Beispiel ist aktuellesJahr eine methodenlokale Variable der Methode getAlter().

Ähnlich den lokalen Variablen sind die Parameter des Konstruktors (den man für diesen Fall als spezielle Methode begreifen kann). Die formalen Parameter einer Methode oder eines Konstruktors gelten für den Rumpf der Methode bzw. des Konstruktors. Ihnen werden beim Aufruf Kopien der Werte der aktuellen Parameter zugewiesen.

Und dann gibt es noch blocklokale Variablen. Z.B. ist in einer typischen For-Schleife

```
for(int i = 0; i < whatever; i++) {
   // Schleifenrumpf
}
```

die Variable i nur innerhalb der For-Schleife gültig, also in deren Schleifenkopf und im Rumpf. Betrachtet man Methoden und Konstruktoren als Blöcke (was legitim ist), so kann man methodenlokale und konstruktorlokale Variablen als Sonderfälle von blocklokalen Variablen ansehen.

Wo können unveränderliche Attribute initialisiert werden?

Frage: Im Beispiel auf Seite 89 steht bzgl. der Zuweisung ax =7 (Zuweisung im Konstruktor): „Korrekt: Initialisierung im Konstruktor und nicht an der Deklarationsstelle.“ Oben im Text steht allerdings: „Unveränderliche Variablen und Attribute müssen entweder an der Deklarationsstelle oder im umfassenden Block bzw. dem Konstruktor initialisiert werden.“ Also könnte die Zuweisung ax=7 auch an der Deklarationsstelle gemacht werden, oder?

Antwort: Ja. Bei Attributen geht beides. Klarer wäre hier: „Unveränderliche blocklokale Variablen (incl. methoden- und konstruktorlokale Variablen) müssen an der Deklarationsstelle initialisiert werden, unveränderliche Attribute entweder an der Deklarationsstelle oder in den Konstruktoren.“

Warum darf eine Klassenmethode nur auf Klassenattribute zugreifen?

Die „normalen Attribute“ sind zwar in der Klasse deklariert (man kann für diesen Fall eine Klasse als Bauplan für die Exemplare ansehen), sie gehören aber zum Zustand der *Exemplare* der Klasse. Jedes Exemplar hat seine eigenen Attribute. Klassenmethoden und -attribute hingegen gehören zur Klasse selbst. Sie gibt es nur *einmal* pro Klasse, und zwar völlig unabhängig davon, ob von dieser Klasse Exemplare überhaupt existieren. Es ergibt also keinen Sinn, aus einer Klassenmethode auf ein „normales Attribut“ zuzugreifen, denn die Klassenmethode gehört zur Klasse und weiß nichts von Exemplaren. Auf *Klassenattribute* hingegen kann sie zugreifen, denn die gehören ebenfalls zur Klasse und sind von Exemplaren völlig unabhängig.

Namen von Objekten / Erzeugung eines Objekts ohne Zuweisung an eine Variable

Welchen Namen hat ein Objekt, das ich keiner Variablen zuweise? Wie kann ich darauf zugreifen?

Objekte haben überhaupt keine Namen. Es gibt lediglich benannte *Variablen*, welche Referenzen auf Objekte enthalten können. Das sollte man sorgfältig auseinanderhalten. Schließlich bleibt ein Objekt immer dasselbe, hingegen kann eine Variable im Laufe ihres Lebens Referenzen auf höchst unterschiedliche Objekte enthalten:

```
Person a = new Person("Michael", "Paap");
Person b = new Person("Gustav", "Gans");
Person x = b;
b = a;
```

Und welche Variable ein Objekt referenziert, welches man z.B. mit

```
new Test(42);
```

erzeugt hat, hängt davon ab, welcher Variable man das Ergebnis dieses Konstruktoraufrufs zuweist. Z.B.:

```
Test einTest = new Test(42);
```

Wenn man die Referenz auf das neu erzeugte Objekt keiner Variablen zuweist, dann wird man später keinen Zugriff mehr auf das Objekt haben. Man kann aber immerhin noch im gleichen Ausdruck auf das Objekt zugreifen. Derlei findet man häufig z.B. bei Threads:

```
new Thread(myRunnable).start();
```

Hier wird der neu erzeugte Thread gestartet, ein solcher Ausdruck ist also keinesfalls folgenlos. Nur der Zugriff auf den Thread ist (jedenfalls auf die übliche Weise) nicht mehr möglich.

Das Schlüsselwort „this“

Kann jemand die Bedeutung von „this“ erklären?

Für „this“ gibt es mehrere Verwendungen. Gemeinsam ist ihnen, dass „this“ das „aktuelle Objekt“ meint. Das ist im Rumpf einer Methode das Objekt, dem die Nachricht geschickt wurde, die zur Ausführung der Methode führte, auch als „impliziter Parameter“ des Mehtodenaufrufs bezeichnet. In einem Konstruktor ist es das Objekt, das gerade
initialisiert wird.

Nun kann man normalerweise Methoden und Attribute des aktuellen Objekts aus diesem selbst heraus abkürzend auch ohne „this“ ansprechen. Es gibt aber Fälle, in denen das „this“ erforderlich ist. Die wichtigsten:

1. Ein Attribut ist durch eine gleichnamige lokale Variable oder einen formalen Parameter verdeckt, man will aber das Attribut ansprechen. Generell sollte man derlei Verdeckungen am Besten vermeiden, es gibt aber einen typischen Fall, der völlig akzeptabel ist, nämlich eine Attributinitialisierung im Konstruktor durch übergebene Parameter:

```
public class Test {

   private int bla;

   private String blubb;

   public Test(int bla, String blubb) {
      this.bla = bla;
      this.blubb = blubb;
   }
}
```

Hier ist das „this“ nötig, damit klar ist, dass bla bzw. blubb auf der linken Seite der Zuweisung das Attribut meint und nicht den gleichnamigen formalen Parameter.

2. Aufruf eines anderen Konstruktors der gleichen Klasse:

```
public class Test {

   private int bla;

   public Test(int bla) {
      this.bla = bla;
      doSomething();
   }

   public Test() {
      this(42);
   }
}
```

Hier wird vom parameterlosen Konstruktor der andere Konstruktor mit einem Default-Wert aufgerufen. Der Vorteil solcher - häufig vorkommenden - Konstrukte ist z.B., dass man Initialisierungen, die allen Konstruktoren gemeinsam sind nur einmal in dem einen Konstruktor schreiben muss, den alle anderen verwenden.

3. Man will einem Methoden- oder Konstruktoraufruf eine Referenz auf das „aktuelle Objekt“ mitgeben, um so dem Empfänger des Aufrufs die „Verwendung“ dieses Objekt zu ermöglichen. Dazu kann man „this“ übergeben. Hier ein (mäßig sinnvolles) Codebeispiel:

```
class Weckdienst {

   private LinkedList<Person> zuWeckendePersonen = new LinkedList<Person>();

   public void anmeldenZumWecken(Person p) {
      zuWeckendePersonen.add(p);
   }

   public void jetztAlleWecken() {
      for (Person p : zuWeckendePersonen) {
         p.wecken();
      }
   }
}
```

Bei einem solchen Weckdienst kann man eine Person durch Aufruf der Methode anmeldenZumWecken() anmelden, wobei man die Person als Parameter übergibt. Nun nehmen wir einmal an, eine Person will *sich selbst* zum Wecken anmelden. Dann muss sie ja eine Referenz auf sich selbst übergeben. Dazu könnte in einer ihrer Methoden das hier stehen:

```
meinWeckDienst.anmeldenZumWecken(this);
```

4. Aus einer (nichtstatischen) inneren Klasse heraus kann man das „umgebende Exemplar“ der äußeren Klasse ansprechen, und zwar über den Klassennamen der äußeren Klasse, gefolgt von einem Punkt und dem Schlüsselwort „this“. Dies ist dann nötig, wenn man Elemente des „umgebenden Exemplars“ ansprechen will, diese aber durch gleichnamige Elemente der inneren Klasse verdeckt werden. Der Fall ist also ähnlich dem unter Punkt 1 genannten.

Kann „this“ jemals null sein?

Nein. Denn das Schlüsselwort „this“ bezeichnet immer das „aktuelle Objekt“, im Falle einer Methode das Objekt, auf dem die Methode aufgerufen wurde, im Falle eines Konstruktors das Objekt, welches gerade erzeugt wird. Das Schlüsselwort null bezeichnet die leere Referenz, also eine Referenz, die kein Objekt referenziert. Da man einem nicht vorhandenen Objekt aber keine Nachricht schicken kann, *kann* this niemals null sein. Deshalb ist der folgende Code absolut sinnlos, da die Prüfung *immer* zu true auswerten wird:

```
void aMethod() {
```

```
    if (this != null) {
        // ...
    }
}
```

Und in einer statischen Methode? Dort müsste doch „this“ gleich null sein, weil es für statische Methoden kein „aktuelles Objekt“ gibt?

Es ist korrekt, dass statische Methoden *nie* auf einem Objekt aufgerufen werden, auch wenn Java (leider) eine Syntax erlaubt, die diesen Eindruck erweckt. Also gibt es keine Nachrichtenempfänger und damit auch kein this. Aber das heißt nicht, dass this in einer statischen Methode den Wert null hat: Der Compiler unterbindet schlicht jegliche Verwendung von this in einem statischen Kontext.

Enum

Ist eine Enum eine Klasse?

Bei „enum“ handelt es sich zunächst einmal um ein Schlüsselwort. Es heißt „enum“, wie es „class“ und „interface“ heißt. Aber ja, eine Enum ist eine spezielle Art von Klasse. Man kann - mit bestimmten Einschränkungen - eine Enum so deklarieren und implementieren, wie man eine Klasse deklariert, insbesondere muss eine solche Enum-Klasse auch keinesfalls eine innere Klasse einer anderen Klasse sein, auch wenn das in Beispielen oft so ist. Und letztlich baut der Compiler aus einer Enum eine normale Klasse mit einer ganz normalen Class-Datei. Aber es gibt auch eine Klasse Enum<E>, die die gemeinsame direkte Superklasse aller Aufzählungstypen bildet.

Kann ich über eine Enum iterieren, z.B. mit der ForEach-Form der For-Schleife?

Nein, nicht direkt. Aber jede Enum-Klasse verfügt implizit immer über die statische Methode values(). Diese liefert ein Array, welches alle „Werte“ des enum liefert. Und über dieses Array kann man dann ganz normal iterieren, auch mit der ForEach-Schleife.

Wo finde ich die Methoden values() und valueOf(String name) in der API-Doku?

Leider überhaupt nicht. Es handelt sich um *implizit deklarierte statische Methoden*, deren Vorhandensein für jede Enum-Klasse man schlicht kennen und akzeptieren muss… einer der unschönen Sonderfälle der Sprache Java.

Problem mit enum in Aufgabe 3 der Nachklausur zum Sommersemester 2007

Frage: In der Aufgabe geht es um eine Aufzählung mittels enum. Dabei kann man einerseits einer Variablen einen Wochentag.Freitag zuweisen, indem man die entsprechende Enum-Konstante verwendet. Später kann man dann aber genau diesen Freitag verändern, z.B. durch Auruf der Methode naechsterTag(). Das hat dann den Effekt, daß der Wochentag zwar irgendwie Wochentag.Freitag ist, aber gleichzeitig wt.aktuellerTag Samstag liefert:

```
Wochentag wt = Wochentag.Freitag;
System.out.println("Wochentag: " + wt);
System.out.println("aktueller Wochentag: " + wt.aktuellerTag);
wt.naechsterTag();
System.out.println("Wochentag: " + wt);
System.out.println("aktueller Wochentag: " + wt.aktuellerTag);
```

Irgendwie scheint mir das irreführend zu sein. Wie und wo würde man denn diese seltsame Aufzählung sinnvoll einsetzen?

Antwort: Da muss ich dir recht geben. Die Aufgabe ist so leider nicht wirklich sinnvoll und du hast auch schon genau erkannt, warum nicht.

Die neue Klasse Wochentag soll ja ein eine Enum-Klasse sein, welche die Wochentage repräsentiert. D.h. ein Exemplar dieser Klasse steht genau für einen Wochentag und kann selbst als Wert einer entsprechenden Variablen aktuellerTag verwendet werden. Die Wochentags-Enum dann noch einmal als einen „Behälter“ für einen Wert, der einen bestimmten Tag repräsentiert, zu gestalten, ergibt damit keinen Sinn.

Man kann aber das Beispiel so umbauen, dass es etwa dem entspricht, was in der Aufgabe eigentlich beabsichtigt war und dabei eine Enum sinnvoll verwendet. Dazu baue ich eine Klasse Tag, die die Rolle des Behälters für einen konkreten Wochentag bekommt, und in die ich die Funktionalität zur Bestimmung des vorigen/nächsten Tags einbaue. Das könnte etwa so aussehen:

```
public class Tag {

   public enum Wochentag {
      Montag, Dienstag, Mittwoch, Donnerstag,
      Freitag, Samstag, Sonntag
   }

   private Wochentag aktTag = Wochentag.Montag;

   public Wochentag getTag() {
      return aktTag;
   }

   public void setTag(Wochentag tag) {
      aktTag = tag;
   }

   public void naechsterTag() {
      aktTag = Wochentag.values()[(aktTag.ordinal() + 1) % 7];
   }

   public void vorhergehenderTag() {
      aktTag = Wochentag.values()[(aktTag.ordinal() + 6) % 7];
   }

   public static void main(String[] args) {
      Tag t = new Tag();
      t.setTag(Wochentag.Dienstag);
      t.naechsterTag();
      System.out.println(t.getTag());
   }
}
```

Schöner ist es natürlich, die Methoden zur Berechnung des nächsten/vorigen Tags gleich in die Enum-Klasse einbauen. Das sollte dann aber so geschehen, dass die Methoden wiederum ein Exemplar der Enum-Klasse liefern. Analog zu obigem Beispiel könnte das so aussehen:

```
public class Tag {

   private Wochentag aktTag = Wochentag.Montag;

   public Wochentag getTag() {
      return aktTag;
   }

   public void setTag(Wochentag tag) {
      aktTag = tag;
   }

   public static void main(String[] args) {
      Tag t = new Tag();
      t.setTag(Wochentag.Dienstag);
      t.setTag(t.getTag().naechsterTag());
      System.out.println(t.getTag());
   }
}

public enum Wochentag {
   Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag;

   public Wochentag naechsterTag() {
      return values()[(ordinal() + 1) % 7];
   }

   public Wochentag vorhergehenderTag() {
      return values()[(ordinal() + 6) % 7];
   }
}
```

Verwendung von static

Wann soll/darf ich static verwenden? Wozu ist static gut?

Kurz gesagt: In einem Programm eines eher unerfahrenen Programmierers sollte am Besten nur eines static sein, und zwar die main-Methode. Ansonsten ist mit hoher Wahrscheinlichkeit davon auszugehen, dass weitgehend prozedural und nicht objektorientiert programmiert wurde. Man sollte sich klar machen, dass statische Elemente im Grunde den globalen Prozeduren und Variablen der prozeduralen Programmierung entsprechen, nur dass sie in Java Klassen zugeordnet sind, wobei diese Zuordnung in erster Linie dazu dient, den Namensraum „thematisch“ zu unterteilen. Prinzipiell ist eine statische Methode oder Variable von überall her zugreifbar, indem man ihrem Bezeichner den Klassennamen voranstellt (eventuelle Sichtbarkeitsmodifikatoren mal außen vor gelassen). Exemplare der betreffenden Klasse sind dazu nicht notwendig.

Leider erlaubt Java es, statische Elemente *scheinbar* über Exemplare der betreffenden Klasse anzusprechen, also z.B. statische Methoden scheinbar auf Exemplaren aufzurufen. In Wirklichkeit werden solche Aufrufe aber vom Compiler in Aufrufe über den Klassennamen umgewandelt. Aus einem

```
Thread t = myThread.currentThread();
```

wird also im Bytecode das gleiche wie bei

```
Thread t = Thread.currentThread();
```

denn die statische Methode currentThread() ist völlig unabhängig von der Existenz irgendeines konkreten Thread-Exemplars.

Man sollte sich unbedingt abgewöhnen, statische Zugriffe „über Exemplare“ durchzuführen, weil es vernebelt, was man da eigentlich tut und damit das Verständnis erschwert. Gute IDEs warnen einen in solchen Fällen mit „The static method currentThread() from the type Thread should be accessed in a static way“ oder ähnlichen Meldungen.

Eine Ursache für den Missbrauch von „static“ liegt wohl in Compilerfehlermeldungen à la „Cannot make a static reference to the non-static ...“. Nehmen wir folgenden Code:

```
public class Motorrad {

   private boolean motorLaeuft;

   public void start() {
      motorLaeuft = true;
   }

   public static void main(String[] args) {
      start();
   }
```

```
}
```

Der Compiler meldet: „Cannot make a static reference to the non-static method start() from the type Motorrad.“ Die Fehlermeldung legt die Vermutung nahe, dass die Lösung darin bestünde, die Methode start() static zu machen. Gesagt, getan. Nun meldet der Compiler einen anderen Fehler: „Cannot make a static reference to the non-static field motorLaeuft“. Und schon wird auch die Variable static gemacht. Und immer fröhlich so weiter. Das ist *FALSCH*.

Einer der Grundgedanken objektorientierter Programmierung ist es, Daten und die auf ihnen möglichen Operationen in einer Struktur zusammenzufassen, einem Objekt. Man kann dem Objekt Nachrichten senden (Methoden aufrufen) und dann tut es etwas, typischerweise verändert es seinen Zustand, welcher durch seine Attributen repräsentiert wird. Wenn ich auf den Starterknopf meines Motorrads drücke, dann rödelt der Anlasser und wenn alles gut geht, ändert sich der Zustand meines Motorrades signifikant. Ich sende also durch den Knopfdruck eine Nachricht an *ein ganz bestimmtes Motorrad* und *dieses eine Motorrad* reagiert darauf.

Dem entspricht in einer ganz primitiven Simulation der Aufruf einer Methode starte() auf einem Motorrad-Exemplar, welches daraufhin ein boolsches Attribut „motorLaeuft“ von false auf true setzt.

Das Attribut motorLaeuft darf natürlich *nicht* static sein, sonst wäre es ein gemeinsames Attribut *aller* Motorräder, d.h. die wären alle gleichzeitig entweder an oder aus. Das will man nicht. Und damit darf auch die Methode starte() nicht static sein, denn eine statische Methode wird *nicht auf einem Exemplar* aufgerufen (hat keinen impliziten Parameter), sie *kennt* überhaupt keine Exemplare.

Kurz gesagt: Wenn ich ein Motorrad starten will, brauche ich auch ein Motorrad, also ein *Exemplar*. Und damit sähe der Code fürs Motorrad sinnvollerweise etwa so aus:

```
public class Motorrad {

   private boolean motorLaeuft;

   public void start() {
      motorLaeuft = true;
   }

   public static void main(String[] args) {
      Motorrad m = new Motorrad();
      m.start();
   }
}
```

Eine weitere missbräuchliche Verwendung von „static“ besteht darin, dass irgendwelche Werte, die in einer anderen Klasse benötigt werden, als der, in der sie deklariert wurden, static gemacht werden, weil man sie dann von überallher mit vorangestelltem Klassennamen ansprechen kann. Das ist im Allgemeinen nicht sinnvoll. Wenn ein Objekt eine Information benötigt, dann sollte man ihm eine Referenz auf das Objekt verschaffen, welches die Information hat, indem man z.B. eine solche Referenz im Konstruktor übergibt. Sehr oft ist aber auch das

Design als solches das Problem, und es wäre besser, den Job, zu dem die Information benötigt wird, eben dort erledigen zu lassen, wo diese vorhanden ist. Wenn man in seltenen Fällen wirklich einmal einen Weg braucht, variable Informationen programmweit verfügbar zu machen, dann kann ein sog. *Singleton* ein möglicher Weg sein, eine Klasse, von der es genau ein Exemplar gibt, welches man sich mit eine statischen Methode der Klasse besorgen kann (das *kann* eine sinnvolle Anwendung von „static“ sein).

Vielleicht sollte ich aber - ohne Anspruch auf Vollständigkeit – auch noch einige Beispiele für sinnvolle Verwendungen des Schlüsselworts „static“ nennen:

1. Da wäre zum einen natürlich die main-Methode. Diese dient als Einstiegspunkt des Programms und ist static, denn sie ist nicht an Exemplare gebunden. Dass sie sich - wie auch in meinem Motorradbeispiel - oft in einer Klasse befindet, von der man auch Exemplare erzeugen will, ist im Grunde reiner Zufall. Man kann sie problemlos in eine andere Klasse auslagern und diese nur dazu nutzen, das Programm zu starten und vielleicht sollte man das auch prinzipiell tun, zumindest so lange, bis einem klar ist, dass es mehr oder weniger nur eine Frage der Optik ist, wo sie steht:

```
public class Start {

   private Start() {}

   public static void main(String[] args) {
      Motorrad m = new Motorrad();
      m.start();
   }
}

public class Motorrad {

   private boolean motorLaeuft;

   public void start() {
      motorLaeuft = true;
   }
}
```

Hier habe ich der Klasse Start extra einen privaten Konstruktor verpasst, damit klar wird, dass von ihr gar keine Exemplare erzeugt werden *können*.

2. Eine weitere sinnvolle Verwendung von „static“ sind reine Dienstleisungsklassen. Diese bündeln Funktionalität, die in prozeduralen Sprachen typischerweise in Form von globalen Funktionen realisiert ist. Ein typisches Beispiel wäre die Klasse Math. Von dieser Klasse sollen und können keine Exemplare erzeugt werden, sie stellt aber Methoden zur Verfügung, die z.B. den Sinus eines übergebenen Wertes zurückliefern. Außerdem enthält die Klasse Math Konstanten, und zwar wieder solche, die in einem prozeduralen Programm globale Konstanten wären, wie z. B. Math.PI oder Math.E. Womit wir eine weitere sinnvolle Verwendung für „static“ haben, nämlich eben solche global gültigen Konstanten.

3. Im Zusammenhang mit Factories findet man oft statische Methoden, so genannte Factory-Methoden. Eine solche Methode wird dann statt eines Konstruktors verwendet, wenn man ein Exemplar eines bestimmten Typs benötigt. Einer der Vorteile ggü. einem Konstruktor besteht darin, dass die Factory-Methode - abhängig von irgendwelchen Faktoren - auch ein Exemplar eines Subtyps des Typs liefern kann, den sie als Rückgabetyp deklariert. Ein Beispiel wäre die Methode getInstance() der abstrakten Klasse Calendar, welche abhängig von der verwendeten Locale und Zeitzone vorkonfigurierte Exemplare einer Subklasse von Calendar liefert.

Aber wenn ich einfach nur Code aus einer Klasse verwenden will…

Oft taucht die Frage auf, wie man eine Methode irgendeines Objektes einer Klasse benutzt, die mit *diesem* Objekt als solchem nichts zu tun hat, sondern einfach gut passender Code ist, der halt in der Klasse von m steht steht. Wenn es nur darum ginge, *dann* könnte man die Methode ja wirklich „static“ machen, weil es dann eine reine Diestleistungsmethode wäre, analog etwa zu Math.pow().

Der Normalfall ist aber, dass man aus einem ganz bestimmten Objekt a einem ganz bestimmten Objekt b eine Nachricht senden will, um von *diesem speziellen Objekt* etwas zu erfahren oder den Zustand *dieses* Objekts zu ändern. Dann benötigt man aber in a eine Referenz auf b. Ich erweitere mein Motorrad-Beispiel…

Disclaimer: Das folgende Beispiel ist konstruiert und wie alle Beispiele mit Bedacht zu genießen. Es trifft eine Reihe impliziter Annahmen, um es sinnvoll erscheinen zu lassen. Es soll natürlich *nicht* zeigen, wie man üblicherweise die Vorgänge der Motorradfabrikation oder der Teilebestellung realisiert. Sowas läuft ganz anders und ist um einige Größenordnungen komplexer. Das Beispiel dient lediglich dazu, auf einem stark vereinfachten Modell bestimmte Problemstrukturen zu verdeutlichen, (die einem in der Realität aber durchaus begegnen) und typische Lösungen zu zeigen.

Also: Ich habe ein „richtiges“ Objekt, welches bestimmte veränderliche Daten hat, also einen Zustand, und Methoden, mit denen man den Zustand abfragen oder verändern kann. Dieses Objekt soll so gestaltet werden, dass es eine dauerhafte Beziehung zu einem anderen Objekt bekommt, das es braucht, um korrekt zu funktionieren. Dann ist eine typisches Struktur die weiter unten gezeigte, wobei ich das jetzt so modelliere, dass lediglich das Motorrad seinen Motor und seinen Anlasser kennt, diese sich aber nicht gegenseitig. Das ist sinnvoll, denn so kann man eins der Teile austauschen und muss das nur *einer* Stelle mitteilen. Ein ganz bestimmter Anlasser und ein ganz bestimmter Motor gehören also zum Objektzustand genau *des* Motorrads, welches ich in der main-Methode erzeuge:

```
public class Start {

  private Start() {}

  public static void main(String[] args) {
     Motorrad m = new Motorrad();
     m.starte();
  }
}
public class Motorrad {
```

```
  private Anlasser anlasser;

  private Motor motor;

  public void starte() {

     anlasse.dreheMotor(motor);
  }
}
```

So... wie bekomme ich denn nun aber den Motor und den Anlasser beim Bau des Motorrades in Beziehung zum Motorrad? Ein gängiger Weg ist die Übergabe im Konstruktor:

```
public class Start {

  private Start() {}

  public static void main(String[] args) {
     Anlasser anl = YamahaZentralLager.
           getAnlasser("XJ900 Diversion", 1995);
     Motor mot = YamahaZentralLager.
           getMotor("XJ900 Diversion", 1995);
     Motorrad meinMotorrad = new Motorrad(anl, mot);
     meinMotorrad.starten();
  }
}

public class Motorrad {

  private Anlasser anlasser;

  private Motor motor;

  public Motorrad(Anlasser anlasser, Motor motor) {
     this.anlasser = anlasser;
     this.motor = motor;
  }

  public void starten() {
     anlasser.dreheMotor(motor);
  }
}
```

Und so baut sich nach und nach ein objektorientiertes Programm auf, in dem das Starten eines Motorrades genau *den* Anlasser *dieses* Motorrades dazu bringt, den Motor *dieses* Motorrades zu drehen. Aber stop... YamahaZentralLager ist offenbar eine Klasse (Damit man das auf den ersten Blick sieht, gibt es die Konvention, Klassen groß zu beginnen, Methoden hingegen klein), und getAnlasser() und getMotor() sind *statische* Methoden. Offenbar gibt es nur ein einziges YamahaZentralLager und da hab ich mir gedacht, ich kann die Methoden ja auch static machen. ;-)

Aber will man das wirklich so bauen? Betrachten wir den Fall genauer. Ich ändere das Beispiel ein bisschen um: Nehmen wir zunächst an, Yamaha hat genau ein Zentrallager in Japan, aus dem Besteller Teile anfordern können, wobei dem Zentrallager mitgeteilt wird, wer der Besteller ist, damit die Ware dorthin ausgeliefert wird. Dann könnte die zu obigem Beispiel passende Klasse YamahaZentralLager etwa so aussehen:

```
public class YamahaZentralLager {

   private static int motorVorrat;

   public static boolean liefereMotor(Besteller b) {
      if (motorVorrat > 0) {
         motorVorrat --;
         spedition.liefereAus(einMotor, b);
         return true;
      }
      return false;
   }
}
```

Und irgendwo in einer Methode eines Bestellers könnte es einen Aufruf geben, der so aussieht:

```
YamahaZentralLager.liefereMotor(this);  // this ist der Besteller
```

Und schon schickt das Zentrallager einen Motor auf die Reise, und zwar zu dem als Parameter übergebenen Besteller. Das entspricht strukturell dem, was ich im ersten Beispiel gemacht habe, nur dass ich im folgenden irrelevante Details wie etwa das Baujahr weggelassen habe und stattdessen den Besteller übergebe.

Schön ist das aber nicht. Stattdessen wird man das Zentrallager eher als Singleton modellieren. Einer der Hauptgründe: Angenommen, irgendwann wird dann *doch* ein weiteres Lager in Europa gebaut. Dann müssen *alle*, die bisher Motoren angefordert haben, ihre Anforderungsmethode ändern. Das mag der Realität in vielen Bereichen der Wirtschaft entsprechen, aber man kann ja beim Modellieren auch einmal etwas besser machen als in der Realität.

Habe ich das Lager von vornherein als Singleton realisiert, dann kann ein Besteller so arbeiten:

```
YamahaZentralLager.getInstance(this).liefereMotor();
```

Und das funktioniert auch noch, wenn es später doch mehrere ZentralLager gibt. Dann kann ich nämlich einfach die Methode getInstance() so umbauen, dass sie je nach Standort des Bestellers *unterschiedliche* ZentralLager-Exemplare liefert. Und auch in der Klasse ZenralLager halten sich die Änderungen in Grenzen. Insbesondere müssen nicht alle Attribute von static auf nichtstatic geändert werden.

Wird static in Java nicht auch verwendet, um Aufzählungstypen zu modellieren?

So hat man das in der Tat vor 1.5 gemacht. Das ist letztlich ein Sonderfall der von mir erwähnten sinnvollen Verwendung von static für global gültige Konstanten. Allerdings hat diese Vorgehensweise reichlich Nachteile, weswegen es seit Java 1.5 glücklicherweise typsichere Aufzählungstypen (enum) gibt. Diese werden im Kurs in Abschnitt 3.2.6.2 behandelt und auch hier in der FAQ gibt es ein Kapitel zu Enums. Eine ausführlichere (über den Kurs hinausgehende) Einführung mit Beispielen findet sich unter http://download.oracle.com/javase/1.5.0/docs/guide/language/enums.html.

Auflösung von Überladung / Überschreiben, dynamisches Binden

Was ist eigentlich dieses „Binden“?

„Binden“ bezeichnet den Vorgang, durch den einem im Quellcode stehenden Methodenaufruf die Methode zugeordnet wird, welche dann tatsächlich ausgeführt wird.

Wie läuft das „Binden“ bei Java ab? Was macht der Compiler und was die VM?

Dieser Vorgang erfolgt bei Java in zwei Stufen.

Zunächst erfolgt die Auflösung von Überladung durch den Compiler ausschließlich aufgrund der Deklarationstypen (statischen Typen) des Ausdrucks, auf dem der Methodenaufruf erfolgt sowie der Parameter des Aufrufs. Als Ergebnis der Tätigkeit des Compilers wird ein Aufruf *einer* ganz bestimmten Methode in den Bytecode geschrieben, oder aber es kommt zu einem Compilerfehler, nämlich dann, wenn der Compiler unter mehreren zum Aufruf passenden Methoden keine Entscheidung treffen kann.

Zur Laufzeit ordnet die Laufzeitumgebung („Virtual Machine“, VM) dem im Bytecode stehenden Methodenaufruf anhand des tatsächlichen (dynamischen) Typs des Objekts, an das sich der Methodenaufruf richtet die letztlich auszuführende Methode zu (dynamische Methodenwahl).

Wie funktioniert die Auflösung der Überladung durch den Compiler genau?

Antwort: Die Auflösung von Überladung durch den Compiler erfolgt nach dem Most-Specific-Algorithmus, der selbst wieder in mehreren Schritten abläuft. Dabei werden prinzipiell nur die Deklarationstypen (= statischen Typen) des Empfängers des Methodenaufrufs sowie der Parameter berücksichtigt, denn nur diese sind dem Compiler bekannt.

Im ersten Schritt wird eine Liste mit zum Aufruf passenden Methoden angelegt, das sind alle, für die gilt, dass die Deklarationstypen der Aufrufparameter Subtypen der Deklarationstypen der formalen Parameter der Methode sind. Besteht diese Liste nur aus einem Element, wird direkt der betreffende Methodenaufruf in den Bytecode geschrieben.

Wenn die Liste der zum Aufruf passenden Methoden mehrere Elemente enthält, ist ein weiterer Schritt erfoderlich, die Auswahl der *speziellsten* Methode. Dazu wird aus der Liste jede Methode gestrichen, für die es in der Liste eine *speziellere* gibt. Der Vergleich, welche von zwei Methoden spezieller ist, ist dabei völlig unabhängig vom konkreten Methodenaufruf: Eine Methode M2 ist genau dann spezieller als eine Methode M1, wenn für jeden formalen Parameter von M2 gilt, dass sein Typ Subtyp des Typs des entsprechenden formalen Parameters von M1 ist. Bleibt nach dem Streichen genau eine Methode übrig, ist diese die speziellste zum Aufruf passende und ein entsprechender Aufruf wird in den Bytecode geschrieben. Bleiben

mehrere Methoden übrig, dann sind diese offenbar gleich speziell, und es ist nicht möglich, den Methodenaufruf eindeutig aufzulösen. → Compilerfehler.

Wie funktioniert das dynamischen Binden durch die Laufzeitumgebung genau?

Als Ergebnis der Tätigkeit des Compilers steht im Bytecode ein konkreter Methodenaufruf. Zur Laufzeit muss nun die Laufzeitumgebung („Virtual Machine“, VM) diesem Methodenaufruf die tatsächlich auszuführende Methode zuordnen.

Diese (dynamische) Zuordnung ist notwendig, weil der tatsächliche Typ (dynamische Typ) des Objekts, an das sich der Methodenaufruf richtet, ein beliebiger Subtyp des Deklarationstyps des Ausdrucks sein kann, auf dem der Methodenaufruf erfolgte. Im dynamischen Typ könnte also die vom Compiler ausgewählte Methode *überschrieben* worden sein, und dann soll ja die Methode des tatsächlichen Typs des Aufrufempfängers ausgeführt werden.

Da in Java eine Methode nur dann eine gleichnamige Methode eines Supertyps überschreibt, wenn die Deklarationstypen der Parameter beider Methoden exakt übereinstimmen (Invarianz), sucht die VM beginnend im tatsächlichen Typ des Aufrufempfängers nach einer Methode, deren Parametertypen *genau* denen des im Bytecode stehenden Methodenaufrufs entsprechen. Findet sie diese Methode dort nicht, schaut sie in der Superklasse nach, ggf. in deren Superklasse und so weiter. *Dass* sich die Methode finden muss, ist klar, sonst hätte der Compiler den entsprechenden Methodenaufruf ja gar nicht erst zugelassen.

Warum ist es falsch, zu sagen, im zweiten Schritt des Most-Specific-Algorithmus würde die „am besten zum Aufruf passende“ Methode ausgewählt?

Antwort: Weil aus der Liste der prinzipiell zum Aufruf passenden Methoden die *speziellste* Methode ausgewählt wird. Dazu wird aus der Liste jede Methode gestrichen, zu der es eine *speziellere* gibt. Die Frage, welche von zwei Methoden spezieller ist, ist aber völlig unabhängig vom einem konkreten Methodenaufruf: Eine Methode M2 ist genau dann spezieller als eine Methode M1, wenn für jeden formalen Parameter von M2 gilt, dass sein Typ Subtyp des Typs des entsprechenden formalen Parameters von M1 ist.

Ein Vergleich zwischen den aktuellen Parametern eines tatsächlichen Methodenaufrufs und den formalen Parametern der verschiedenen prinzipiell passenden Methoden („passendste Methode“) kann zu einem anderen Ergebnis führen als die oben beschriebene tatsächliche Vorgehensweise („speziellste passende Methode“) und ist daher falsch.

Wieso sind im folgenden Beispiel die beiden Methoden „gleich speziell“, obwohl doch der „Abstand“ zwischen Karpfen und Fisch viel geringer ist als der zwischen Tier und Object?

Gegeben war die folgende Typhierarchie:
Karpfen SUB→ Fisch SUB→ Tier SUB→ Lebewesen SUB→ Object.

Es ging um diese beiden Methoden:

```
void m1(Karpfen k, Object o) { … }

void m2(Fisch f, Tier t) { … }
```

Antwort: In der Tat… zwischen Karpfen und Fisch liegt ein „Typschritt“, zwischen Tier und Object liegen zwei. Man könnte daher annehmen, dass die obere Methode spezieller sei als die untere. Das ist aber nicht der Fall: Bei der Frage, ob eine Methode spezieller ist, als eine andere spielt nur die *Existenz* von Subtypbeziehungen zwischen den Deklarationstypen der formalen Parameter eine Rolle. Ob und wie viele „Schritte“ zwischen zwei Typen liegen, ist dabei irrelevant.

Für keine der beiden obigen Methoden kann man sagen, dass die Deklarationstypen sämtlicher formaler Parameter Subtypen des entsprechenden Parametertyps der anderen Methode sind: Es gilt ja Karpfen SUB→ Fisch, aber nicht Object SUB→ Tier und es gilt Tier SUB→ Object aber nicht Fisch → Karpfen. Die beiden Methoden sind also gleich speziell.

Kämen also nur diese beiden Methoden für einen bestimmten Aufruf als passende Methoden in Frage oder gäbe es unter den anderen in Frage kommenden Methoden keine, die spezieller ist als beide, wäre der Aufruf nicht auflösbar.

Warum wird im folgenden Beispiel nicht die Methode ausgeführt, die „3“ ausgibt? Schließlich passt diese doch am besten zum Aufruf?

```
public class Test {
   public static void main(String[] args) {
      Vogel v = new Vogel();
      Tier t = new Tier();
      Super sup = new Sub();
      sup.m(v, t);
   }
}

class Super {
   public void m(Tier t1, Tier t2) {
      System.out.println("1");
   }

   public void m(Vogel v1, Vogel v2) {
      System.out.println("2");
   }
}

class Sub extends Super {
   public void m(Vogel v, Tier t) {
      System.out.println("3");
   }
}
```

```
class Tier { }

class Vogel extends Tier { }
```

Antwort: Die Auflösung von Überladung findet *ausschließlich und abschließend* durch den *Compiler* statt. Und der kennt nur die statischen Typen (= Deklarationstypen) von Variablen und Ausdrücken. Der Deklarationstyp der Variablen sup ist aber der Typ Super. Ob dieser Variablen zur Laufzeit eine Referenz auf ein Sub-Exemplar zugewiesen werden wird, weiß der Compiler nicht: Selbst die Zulässigkeit der Zuweisung überprüft er nur anhand der Deklarationstypen (das Resultat eines Konstruktoraufrufs hat ja definitionsgemäß den Deklarationstyp der Klasse, deren Konstruktor aufgerufen wurde). Die Zuweisung ist zulässig, denn es gilt Sub SUB→ Super. Anschließend spielt der Typ Sub für den Compiler keine Rolle mehr, ihn interessieren ja nur Deklarationstypen!

Einschub: Der Grund dafür ist, dass es so dem Compiler erspart bleibt, zu analysieren, was einer Variable im Laufe ihrer Existenz so zugewiesen wird. Das kann ja, wenn es verschiedene mögliche Abläufe gibt, schnell sehr aufwändig werden. Und wenn es sich bei der Variablen nicht um eine methodenlokale Variable handelt, sondern um ein Attribut und man mehrere Threads hat, die Zugriff auf dieses haben, dann ist es oft faktisch unmöglich, vorherzusagen, welches Objekt die Variable zu einem bestimmten Zeitpunkt referenzieren wird.

Und das heißt, dass für den Compiler bei Methodenaufrufen auf der Variablen sup nur *die* Methoden eine Rolle spielen, welche für deren Deklarationstyp Super deklariert sind (in diesem selbst oder von einer seiner Supertypen geerbt). Der Compiler wählt unter diesen Methoden diejenigen aus, welche zum Methodenaufruf passen. Das sind genau die Methoden, für die gilt, dass die Deklarationstypen aller Aufrufparameter Subtypen der entsprechenden formalen Parameter in der Methodendeklaration sind. In unserem Beispiel ist das nur bei der Methode „1“ der Fall. In den Bytecode wird also ein Methodenaufruf von `m(Tier, Tier)` geschrieben.

Zur Laufzeit könnte nun doch noch eine im Typ Sub deklarierte Methode eine Rolle spielen: Da die Variable sup zur Laufzeit ein Sub-Exemplar referenziert, beginnt die VM mit ihrer Suche nach der Methode, deren Aufruf im Bytecode steht, in der Klasse Sub. Wenn also in Sub eine Methode existierte, welche die Methode `m(Tier, Tier)` aus Super überschriebe, dann (und nur dann) würde diese überschreibende Methode zur Ausführung gelangen (dynamische Methodenwahl). Das ist aber in unserem Beispiel nicht der Fall, da die Methode `m(Vogel, Tier)` aus Sub die Methode `m(Tier, Tier)` aus Super ja nicht überschreibt.

Noch einmal ganz deutlich: Das Thema „Überladung“ ist mit der Tätigkeit des Compilers *abgeschlossen*. Eine Suche im Subtyp nach einer Methode, die „besser zum Aufruf passen würde“ als die, deren Aufruf der Compiler in den Bytecode geschrieben hat, findet zur Laufzeit also *nicht* statt.

<u>Bisher war bzgl. des „Bindens“ nur die Rede von Methoden. Man sagt aber auch, in Java würden „Attribute statisch gebunden“. Was heißt das?</u>

Richtig, der Begriff wird auch bei Attributen verwendet. Hier meint er – analog zur Begriffsverwendung bei Methoden – den Vorgang, bei dem einer Attributselektion im Quellcode das Att-

ribut zugeordnet wird, das tatsächlich angesprochen wird. Die Aussage, dass in Java Attribute statisch gebunden werden, bedeutet, dass bei diesem Vorgang nur der Deklarationstyp (= statischer Typ) des Ausdrucks, über den die Attributselektion erfolgt, eine Rolle spielt. Hier ein Beispiel, das die statische Bindung von Attributen demonstriert und dieser die dynamische Bindung bei (nichtstatischen) Methoden gegenüberstellt:

```
public class Test {
   public static void main(String[] args) {
      Super sup = new Sub();
      System.out.println(sup.a);
      System.out.println(sup.m());
   }
}

class Super {
   String a = "Super-Attribut";

   String m() {
      return "Super-Methode";
   }
}

class Sub extends Super {
   String a = "Sub-Attribut";

   String m() {
      return "Sub-Methode";
   }
}
```

Die Ausgabe dieses Programms lautet:

```
Super-Attribut
Sub-Methode
```

Denn der Deklarationstyp der Variablen sup ist Super, also wird die Variable a des Typs Super an die Attributselektion „gebunden“. Hingegen wird bei der Bindung der Methode durch die VM der Laufzeittyp (= dynamischer Typ) des von sup referenzierten Objekts berücksichtigt. Dies ist der Typ Sub. Und da die Methode m() aus Super in Sub überschrieben wurde, wird die Methode m() von Sub ausgeführt („dynamisches Binden“).

Kovarianz und Kontravarianz

Versuch einer ausführlicheren Erläuterung

Die Frage, bei der die Begriffe Ko-/Kontravarianz eine Rolle spielen, ist die, inwieweit eine Methode eines Subtyps, welche eine in einem Supertyp bereits deklarierte Methode überschreibt (bzw. implementiert) sich von der des Supertyps bzgl. bestimmter Aspekte unterscheiden darf, nämlich in Bezug auf den Rückgabetyp, die Typen der formalen Parameter und ggf. die Typen deklarierter Checked Exceptions.

Dabei bedeutet Kovarianz in etwa „gleichgerichtete Abweichung“, Kontravarianz „entgegengerichtete Abweichung“. Der Bezug ist dabei immer das Typverhältnis zwischen Supertyp und Subtyp, d.h. Kovarianz bzgl. irgendeines Faktors liegt vor, wenn die Typbeziehung zwischen den Ausprägungen dieses Faktors in den beiden Typen gleichgerichtet zu der Typbeziehung der Typen selbst verläuft, Kontravarianz, wenn sie entgegengerichtet verläuft.

Beispiel: Überschreibt eine Methode im Subtyp eine Methode des Supertyps so, dass der Rückgabetyp der Subtypmethode Subtyp des Rückgabetyps der Supertypmethode ist, dann sagt man: Im Subtyp wurde die Methode mit kovariantem Rückgabetyp überschrieben. Überschreibt eine Methode im Subtyp eine Methode des Supertyps so, dass der Parametertyp der Subtypmethode Supertyp des Rückgabetyps der Supertypmethode ist, dann sagt man: Im Subtyp wurde die Methode mit kontravariantem Parametertyp überschrieben.

Ganz wichtig also: Wenn man von Ko- oder Kontravarianz spricht, muss man immer dazusagen, auf welchen Faktor man sich überhaupt bezieht. Aussagen wie „Java erlaubt keine Kontravarianz“ oder „Kovarianz wird durch das LSP verboten“ sind daher völlig sinnlos.

So... nach dieser einleitenden Begriffsklärung nun zum eigentlichen Thema.

Anmerkung: Zu beachten ist, dass die folgenden Ausführungen sich nicht auf Java beziehen, sondern auf das Konzept Subtyping als solches. Java ist hier restriktiver, als es theoretisch nötig wäre. Der Kurstext hierzu:

„Java lässt Kontravarianz bei den Parametertypen nicht zu, da dies zu Problemen im Zusammenhang mit überladenen Methodennamen führen würde (vgl. Abschn. 2.1.4, S. 89, vgl. Abschn. 3.3.5, S. 245). Ab der Version 5.0 lässt Java aber Kovarianz bei den Ergebnistypen zu, was in früheren Versionen noch nicht erlaubt war.“

Die konzeptionelle Grundlage, von der wir ausgehen, ist die Definition der Subtypbeziehung, wonach gewährleistet sein muss, dass ein Exemplar eines Subtyps, da es ein Exemplar des Supertyps *ist*, überall dort stehen kann, wo ein Exemplar des Supertyps erlaubt ist (Liskovsches Substitutionsprinzip, im Folgenden abgekürzt LSP, Barbara Liskov, 1988).

Aus dieser Ersetzbarkeitsforderung ergibt sich, dass ein Subtyp die Fähigkeiten des Supertyps, bestimmte Nachrichten zu verarbeiten, nicht einschränken darf. Also darf schon rein konzeptionell eine im Supertyp vorhandene Methode im Subtyp nie durch eine ersetzt werden, welche

- einen Parameter nicht „verträgt“, den die Supertypmethode verträgt,

- deklariert, abrupt mit einer Ausnahme enden zu können, mit der nicht auch die Supertyp-Methode hätte terminieren können, oder
- einen Rückgabewert liefert, den nicht auch die Supertyp-Methode hätte liefern können.

Eine überschreibende Methode darf also die möglichen Parametertypen nicht einschränken, indem sie nur bestimmte Subtypen der in der von ihr überschriebenen Supertypmethode als Parameter erlaubt (das wäre Kovarianz bzgl. der Parametertypen). Hingegen ist es konzeptionell unproblematisch, wenn sie auch Supertypen der Parametertypen der überschriebenen Methode zulässt (Kontravarianz bzgl. der Parametertypen), denn damit verträgt sie immer noch alle Parameter, welche die überschriebene Methode vertragen hat.

Eine überschreibende Methode darf auch nicht deklarieren, einen Supertyp des Rückgabewerts der überschriebenen Methode zu liefern (das wäre Kontravarianz bzgl. des Rückgabewerts). Denn dann könnte sie auch solche Typen zurückgeben, mit denen der Aufrufer der Methode nicht rechnen musste. Hingegen spricht nichts dagegen, wenn sie deklariert, nur bestimmte Subtypen zu liefern (Kovarianz bzgl. des Rückgabewerts). Denn damit liefert sie ggf. nichts, was nicht auch die überschriebene Methode hätte liefern können.

Wichtig ist, zu verstehen, dass und warum Überschreiben einer Methode mit kontravariantem Rückgabetyp oder mit kovariantem Parametertyp *prinzipiell* nicht möglich ist, wenn man die Forderung aufrechterhält, dass Objekte eines Subtyps überall stehen können müssen, wo Objekte des Supertyps erlaubt sind, wohingegen Überschreiben einer Methode mit kovariantem Rückgabetyp und/oder kontravariantem Parametertyp mit dieser Forderung durchaus verträglich ist. Das ist also keine Eigenschaft einer bestimmten Programmiersprache, sondern eine logische Folgerung einer bestimmten – der gängigen – Definition der Subtypbeziehung.

Und nun das Ganze noch mal an zwei halbwegs konkreten Beispielen...

WARNUNG: Die Beispiele dienen nur der Erläuterung des Konzepts. Es handelt sich also trotz der Java-Syntax *nicht* um sinnvollen Java-Code! Insbesondere ist zu beachten, dass Beispiel 1 in Java zwar kompilierbar wäre, allerdings würde durch die Methode

```
Waerme verbrenne(Brennstoff b)
```

in KohleUndOelOfen die Methode

```
Waerme verbrenne(Kohle k)
```

aus KohleOfen *nicht* überschrieben, sondern *überladen*, denn Java erlaubt aus den o.g. logischen Gründen kein Überschreiben einer Methode mit kontravarianten Parametertypen.

Beispiel 2 wäre in Java erst ab 1.5 / 5.0 kompilierbar, da Java Überschreiben einer Methode mit kovariantem Rückgabetyp erst ab dieser Version unterstützt.

Beispiel 1 (Kontravarianz bzgl. Parametertyp):

```
class Brennstoff {
```

```
}

class Waerme {
}

class Kohle extends Brennstoff {
}

class Oel extends Brennstoff {
}

class KohleOfen {
   Waerme verbrenne(Kohle k) {
      // verheize Kohle auf korrekte Art und Weise erzeuge ein Waerme-Exemplar
      return new Waerme();
   }
}

class KohleUndOelOfen extends KohleOfen {
   Waerme verbrenne(Brennstoff b) {
      if (b instanceof Kohle)
         // verheizeKohle wie bisher erzeuge ein Waerme-Exemplar
         return new Waerme();
      else
         // verheize Oel auf korrekte Art und Weise erzeuge ein Waerme-Exemplar
         return new Waerme();
   }
}
```

Hier überschreibt (noch einmal: nicht in Java!) die Methode

```
Waerme verbrenne(Brennstoff b) { ... }
```

der Subklasse die Methode

```
Waerme verbrenne(Kohle k) { ... }
```

der Superklasse. Dies ist konzeptionell möglich, da die überschreibende Methode nach wie vor auch Kohle (als Subtyp von Brennstoff) verarbeiten kann, denn Kohle *ist ein* Brennstoff.

Wenn A SUB→ B bedeutet, A ist Subtyp von B, und ich unter jede Klasse den Parameter ihrer verbrenne()-Methode schreibe, dann sieht die „ist Subtyp von“-Beziehung hier so aus:

Klasse:	KohleOfen	←SUB	KohleUndOelOfen
Parameter:	Kohle	SUB→	Brennstoff

Diese *Gegenläufigkeit* der Supertyp-Subtyp-Beziehungen ist ein Beispiel für Kontravarianz.

Hingegen wäre es nicht möglich, dass eine fiktive Subklasse SteinkohleOfen die Superklassenmethode verbrenne() mit kovariantem Parametertyp überschreibt, etwa so:

```
Waerme verbrenne(Steinkohle k) { ... }
```

Denn dann würde die neue Methode im Gegensatz zu der, welche sie überschreibt, z.B. keine Braunkohle mehr vertragen, womit ein solcher SteinkohleOfen nicht mehr überall eingesetzt werden könnte, wo ein Ofen eingesetzt werden kann → Widerspruch zum LSP.

Beispiel 2 (Kovarianz bzgl. Rückgabetyp):

```
class SuessGebaeck {}

class Mehl {}

class Eier {}

class Milch {}

class Kuchen extends SuessGebaeck {}

class Baecker {
   SuessGebaeck backe(Mehl me, Eier e, Milch mi) {
      // tu irgendwas was ein Gebaeck erzeugt
      return new Gebaeck();
   }
}

class KuchenBaecker extends Baecker {
   Kuchen backe(Mehl me, Eier e, Milch mi) {
      // tu irgendwas was einen Kuchen erzeugt
      return new Kuchen();
   }
}
```

Hier handelt es sich um Überschreiben mit Kovarianz bzgl. des Rückgabetyps: Die Methode

```
Kuchen backe(Mehl m, Eier e, Milch m) { ... }
```

der Subklasse überschreibt die Methode

```
SuessGebaeck backe(Mehl m, Eier e, Milch m) { ... }
```

der Superklasse. Dies ist konzeptionell möglich, da sie nach wie vor ein SuessGebaeck zurückliefert, denn ein Kuchen *ist ein* SuessGebaeck.

Wenn A SUB→ B bedeutet, A ist Subtyp von B, und ich unter jede Klasse den Rückgabetyp ihrer backe()-Methode schreibe, dann sieht die „ist Subtyp von“-Beziehung hier so aus:

Klasse:	Baecker	←SUB	KuchenBaecker
Rückgabetyp:	SuessGebaeck	←SUB	Kuchen

Diese *Gleichläufigkeit* der Supertyp-Subtyp-Beziehungen ist ein Beispiel für Kovarianz.

Hingegen wäre es nicht möglich, dass eine fiktive Subklasse AllesMoeglicheBaecker von Baecker die Superklassenmethode backe() mit kontravariantem Rückgabetyp überschreibt, etwa so:

```
Backprodukt backe(Mehl m, Eier e, Milch m) { ... }
```

Denn dann könnte die neue Methode im Gegensatz zu der, welche sie überschreibt, z.B. auch Brot zurückliefern, welches aber kein SuessGebaeck ist. Ein Verwender der Methode rechnet aber mit SuessGebaeck. Damit könnte ein AllesMoeglicheBaecker nicht mehr überall eingesetzt werden könnte, wo ein Baecker eingesetzt werden kann → Widerspruch zum LSP.

Threads

Wer führt eigentlich welchen Code in einem Java-Programm aus? (Eisenbahnbeispiel 1)

Anlass der Frage war folgender Code:

```
public class Warum {
   public static void main(String[] argv) {
      Test t = new Test();
      t.start();
      t.dotry();
      t.stop();
   }
}

class Test extends Thread {
   public void run() {
      while (true) {
         System.out.println("Hallo, ich komme.");
         try {
            Thread.sleep(500);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

   // Zum Beenden Enter-Taste druecken
   void dotry() {
      try {
         while (System.in.read() == 0) { }
      } catch (Exception e) {
         e.printStackTrace();
      }
      System.out.println("Ich gehe. Auf Wiedersehen!");
   }
}
```

Antwort: Das ist eine sehr interessante Frage. Man muss unterscheiden zwischen dem eigentlichen Thread, der ja eine mehr oder weniger abstrakte Funktionalität des Betriebssystems darstellt, und dem Exemplar der Klasse Thread, welches ja lediglich eine objektorientierte Abstraktion des eigentlichen Threads darstellt, die dazu dient, diesen von Java aus anzusprechen und zu beeinflussen.

In einer idealen OOP-Welt wären die Objekte selbst aktiv, d.h. sie würden prinzipiell parallel agieren und interagieren. Die Realität heutiger OOP-Sprachen sieht aber anders aus: Objekte sind mehr oder weniger passive Gebilde. Sie bündeln Daten mit den auf diesen Daten möglichen Operationen, aber sie sind nicht aktiv. Das Objekt sitzt mit all seinen tollen Methoden herum, und wartet, dass „jemand“ diese ausführt.

Ich verwende gerne folgende (natürlich auch nur begrenzt taugliche) Analogie: Threads sind die Züge, die ein Schienennetz befahren, welches aus den Objekten und ihren Methoden besteht, oder in einer etwas anderen Sichtweise: aus dem Quellcode.

Und auch ein Thread-Exemplar ist ein ganz normales Objekt, mit Methoden wie jedes andere, also ein Teil des Schienennetzes. Dass einige der Methoden dazu führen, dass ein BS-Thread (ein Zug in meinem Bild) gestartet, pausiert oder beendet wird, tut dabei zunächst einmal wenig zur Sache.

Betrachten wir jetzt den fraglichen Quellcode im Lichte dieses Bildes. Zunächst befinden wir uns in der main()-Methode einer Klasse. Das Programm wurde über diese main()-Methode gestartet. Damit haben wir zwangsläufig jemanden, der Code ausführt, also einen Thread, in unserem Bild einen Zug. Den nennen wir mal Z-M (für Main). Z-M fährt nun die main()-Methode ab und kommt zu folgender Zeile:

```
Test t = new Test();
```

Es wird also ein Exemplar der Klasse Test erzeugt, die von der Klasse Thread abgeleitet ist. Damit haben wir ein „stinknormales Objekt“. Ein neuer Zug gelangt hier noch nicht aufs Schienennetz.

```
t.start();
```

Aber nun! Diese spezielle Anweisung erzeugt tatsächlich einen neuen Zug, einen neuen BS-Thread. Nennen wir diesen Z-2. Z-2 wird automatisch auf den Beginn der run()-Methode des von t referenzierten Thread-Exemplars gesetzt und fährt, sobald der Scheduler ihm Rechenzeit gibt, unabhängig von Z-M los und arbeitet den Code der run()-Methode ab. In unserem Falle heißt das, dass er fröhlich im Kreise fährt. Aber was macht in der Zwischenzeit Z-M? Der fährt ja weiter, nämlich zu dieser Zeile:

```
t.dotry();
```

Und hier wird eine Methode eines Objekts aufgerufen. Dass dieses vom Typ Thread ist, ist dabei völlig egal: Z-M befährt die Methode dotry() des von t referenzierten Objekts:

```
try {
   while (System.in.read() == 0) { }
} catch (Exception e) {
   e.printStackTrace();
}
```

Das hier ist nicht sonderlich schön... der Vergleich ist, ebenso wie die Schleife im Grunde überflüssig, weil die read()-Methode ohnehin so lange blockiert, bis an der Konsole jemand die Return-Taste drückt. Ein einfaches:

```
try {
   System.in.read();
} catch (Exception e) {
   e.printStackTrace();
}
```

hätte es also auch getan. Aber wie auch immer... die read()-Anweisung funktioniert wie ein Stopp-Signal: Unser Z-M-Zug steht so lange, bis eine Zeile von der Konsole gelesen werden kann, also Return gedrückt wurde. Währenddessen befährt Z-2 fröhlich die Schleife in der run-Methode des ihn repräsentierenden Thread-Exemplars.

Wenn nun Return gedrückt wird, fährt Z-M wieder los (sobald er vom Scheduler Rechenzeit bekommt), es wird die Abschiedszeile ausgegeben, und Z-M kehrt zu der Stelle zurück, wo er in die Methode dotry() des Thread-Exemplars abgebogen ist, und gelangt nun
zu dieser Zeile:

```
t.stop();
```

Das heißt, Z-M biegt in die stop()-Methode des von t referenzierten Objekts ein. Auch hier ist für Z-M wieder egal, dass es sich bei dem Objekt um eins vom Typ Thread handelt, er sieht stop() als ganz normales Stück Schiene an.

Aber das von t referenzierte Objekt ist halt doch etwas Spezielles, denn es repräsentiert einen BS-Thread, nämlich Z-2. Und seine Methode stop() hat genau den Zweck, dafür zu sorgen, dass dieser BS-Thread Z-2 beendet wird. Was dann auch geschieht. *PLOPP*. Der Zug Z-2 löst sich in Wohlgefallen auf. Das von t referenzierte Objekt allerdings existiert übrigens fröhlich weiter, wie es ja auch schon existierte, als Z-2 noch gar nicht in der Welt war.

Nach Beendigung der stop()-Methode kehrt Z-M wieder zur Aufrufstelle zurück. Aber dort ist die main()-Methode zu Ende... keine Schienen mehr. Und damit löst sich auch Z-M in Wohlgefallen auf. Und da Z-M der letzte Thread war, beendet sich die VM und das Programm ist zu Ende.

Wie funktioniert „synchronized“? (Eisenbahnbeispiel 2)

Antwort: Stellen wir uns vor, wir haben wieder unsere Thread-Züge, die fröhlich irgendwelche Methoden befahren. Bestimmte Codebereiche sind nun mit Sperrsignalen versehen. Fährt ein Zug in einen solchen Bereich ein, wird hinter ihm das Signal auf rot gestellt. Verlässt der Zug am anderen Ende den Bereich, für den das Signal zuständig ist, wird das Signal wieder auf grün gestellt, z.B. per Funk.

Dieser Mechanismus würde genügen, um zu gewährleisten, dass immer nur ein Zug einen so gesicherten Bereich befährt. Aber wir wollen mehr, und nun wird die Eisenbahn-Analogie leider etwas fragwürdig:

Wir wollen nämlich, dass in mehreren voneinander unabhängigen Streckenabschnitten immer nur *insgesamt* ein Zug unterwegs ist. D.h., wenn ein Zug in den Abschnitt A einfährt, soll nicht nur das Signal für Abschnitt A auf rot gehen, sondern auch das für einen weiteren Abschnitt B (und umgekehrt). Und erst, wenn der Zug den Abschnitt A verlässt, sollen beide Signale wieder grün werden.

Das wird nun - weil zu komplex - nicht mehr über direkte Funkschaltungen zwischen Gleiskontakten und Signalen realisiert. Stattdessen kennt jeder Streckenabschnitt eine spezielle Funk-Kontrollstation, mit der seine Einfahrtsignale gekoppelt sind, nämlich so, dass die Kontrollstation zwei Zustände rot und grün hat, und die Signale des Streckenabschnittes sich nach dem Zustand der Kontrollstation richten. Der Trick ist nun, dass diese Kontrollstation für mehrere unabhängige Streckenabschnitte dieselbe sein kann! Wenn nun ein Zug in Abschnitt A einfährt, welchem die Kontrollstation X zugeordnet ist, dann wird die Kontrollstation in den Zustand „rot“ geschaltet. Und damit wird dann nicht nur das Einfahrtsignal für Streckenabschnitt A rot, sondern auch das für alle anderen Streckenabschnitte, die ebenfalls an die Kontrollstation X gekoppelt sind.

Die Kontrollstationen entsprechen natürlich den Objekten, die zur Synchronisation verwendet werden. Bei einer als synchronized gekennzeichneten Methode ist das „this“, bei Blöcken das explizit (über eine Referenz) angegebene Objekt. Die Rolle einer Kontrollstation kann dabei *jedes* Objekt übernehmen. Ein direkter Zusammenhang zwischen dem zur Synchronisation verwendeten Objekt und irgendwelchen Objekten oder Variablen auf die die synchronisierten Codeabschnitte zugreifen, besteht dabei nicht. Oft ist es natürlich naheliegend, genau die gemeinsam verwendete Ressource, auf die man den Zugriff regeln will, auch als „Kontrollstation“ zu verwenden, denn mit ihr hat man immerhin schon ein Objekt, auf das alle betroffenen Codeabschnitte sowieso schon eine Referenz haben. Aber jede solche Konstruktion lässt sich durch eine ersetzen, bei der stattdessen ein anderes - meist extra erzeugtes - Objekt als „Kontrollstation“ verwendet wird.

Als Kontrollstation kommen übrigens nur Objekte in Frage, also keine Primitivtypen. Wenn man also konkurrierende Zugriffe auf eine Primitivtyp-Variable ausschließen will, dann muss man für alle Codeblöcke, welche einen solchen Zugriff durchführen können, ein Objekt finden oder eigens erzeugen, welches für sie als „Kontrollstation“ dient. Dazu muss es natürlich an der Stelle, an der mit Hilfe dieses Objekt synchronisiert werden soll, möglich sein, eine Referenz auf das Objekt zu erhalten.

<u>Wie erklärt das „Eisenbahnbeispiel 2“, dass ein Thread einen Monitor auch mehrfach sperren kann?</u>

Der Sinn der Erlaubnis der mehrfachen Sperrung eines Monitors ist ja, zu ermöglichen, dass ein Thread aus einem Codebereich, der über ein bestimmtes Objekt synchronisiert wurde, weitere Codebereiche betreten kann, die über dasselbe Objekt synchronisiert wurden. Ein typisches Beispiel wäre ein Objekt, dessen sämtliche Methoden als synchronized gekennzeichnet sind: Gäbe es nicht die Möglichkeit der Mehrfachsperrung, dann wäre es nicht möglich, aus einer solchen Methode ein andere Methode desselben Objekts aufzurufen.

Wir hatten ja gesagt, dass durch das Einfahren eines Zuges in einen Abschnitt, welchem eine bestimmte Kontrollstation zugeordnet ist, die Einfahrtsignale für *alle* Streckenabschnitte, welche über diese Kontrollstation überwacht sind, auf rot geschaltet werden. Damit könnte besagter Zug aber eben auch nicht in einen anderen Bereich einfahren, der durch dieselbe Kontrollstation kontrolliert wird, die er ja selbst gesperrt hat. D.h., wir müssen das Modell ergänzen (wodurch es leider noch realitätsferner wird):

Um die mögliche Mehrfachsperrung in das Eisenbahnmodell einzubauen, müssen wir zusätzlich postulieren, dass diese Rotsignale für genau einen Zug nicht gelten, nämlich für den, dessen Einfahrt die Rot-Schaltung überhaupt erst ausgelöst hat. Dieser Zug darf, nachdem er einmal einen den von einer bestimmten Kontrollstation überwachten Bereich eingefahren ist, alle Rotsignale ignorieren, die zu eben dieser Kontrollstation gehören.

„Schützt“ Synchronisation das Objekt, auf dem synchronisiert wird vor parallelen Zugriffen?

Das *kann* so sein. Nämlich dann, wenn alle Methoden eines Objekts, die es erlauben, dessen Attribute zu ändern, als synchronized markiert sind und ein Zugriff auf Attribute auch nur über diese Methoden möglich ist. Aber allgemein ist dies nicht der Fall.

Nehmen wir an, der Zugriff auf eine Ressource X ist *nur* über bestimmte bekannte Codeblöcke möglich. Dann kann man verhindern, dass konkurrierender Zugriff auf X erfolgt, indem man all diese Codeblöcke synchronisiert und dabei zur Synchronisation immer dasselbe Objekt verwendet. Das *kann* die Ressource X selbst sein, muss es aber nicht.

Lesetipps

Als Ergänzung zum Kurs

Leseempfehlungen sind immer so eine Sache... zu verschieden sind die Vorlieben und die Vorkenntnisse, zu unterschiedlich die Herangehensweisen. Daher: Die unten aufgelisteten Quellen sind meine völlig subjektiven Empfehlungen, wobei ich neben meinem eigenen Eindruck auch Erfahrungen von Belegern des Kurses 1618 aus den letzten Jahren berücksichtigt habe.

Ich möchte aber hiermit ganz deutlich darauf hinweisen, dass kein Buch das *Bearbeiten des Kurses* ersetzen kann. Dieser ist eben*kein* Java-Kurs, sondern behandelt neben der Sprache Java auch Konzepte der objektorientierten Programmierung, und zwar in einer Tiefe, die keines der unten genannten Bücher erreicht. Die im Folgenden genannten Bücher können aber insbesondere denen, die Probleme beim praktischen Umgang mit der Programmiersprache Java haben, eine Hilfe sein.

- **Head First Java (englisch)**

  Meine absolute Empfehlung für jeden, der sich - was sehr zu empfehlen ist - praktisch mit Java beschäftigen will. Der erfrischende Stil der Head-First-Reihe ist sicherlich Geschmackssache, aber ich finde ihn einfach prima. Außerdem behandelt das auch konzeptionelle Fragen der objektorientierten Programmierung, wenn auch aus dem Java-Blickwinkel und natürlich nicht in der Tiefe, in der der Kurs das tut. Das Taschenbuch kostet 31,95 € und ist z.B. bei Amazon erhältlich: Head First Java

- **Java von Kopf bis Fuß**

  Die deutsche Übersetzung von „Head First Java“. Steht der englischen Fassung qualitativ nicht nach, aber ich bevorzuge denoch das Original. Das broschierte Buch kostet 49,90 € und ist z.B. bei Amazon erhältlich: Java von Kopf bis Fuß

- **Handbuch der JavaProgrammierung von Guido Krüger, 6. Auflage**

  Nicht unbedingt für Programmier-Neulinge geeignet, aber wenn man bereits Programmiererfahrung hat, ein sehr gutes Nachschlagewerk zum Arbeiten mit Java, vernünftig gegliedert und mit vielen Codebeispielen. Eine HTML-Version kann unter http://www.javabuch.de kostenlos heruntergeladen werden. Das gebundene Buch kostet 49,80 € und ist z.B. bei Amazon erhältlich: Handbuch der Java-Programmierung: Standard Edition Version 6, m. DVD-ROM

- **The Java ™ SE Tutorials von Oracle (früher Sun), englisch**

  Sozusagen das „Orginal“, jedenfalls von den Leuten, die es eigentlich wissen müssten. Gut geschrieben, für den praktisch Interessierten mit Englisch-Kenntnissen, praktisch orientiert und teilweise deutlich über den Kurs hinausgehend, was aber z.B. unter dem Aspekt einer späteren Teilnahme am Programmierpraktikum sicher kein Fehler ist. Meines Erachtens ist die unter http://download.oracle.com/javase/tutorial verfügbare kostenlose Online-Version völlig ausreichend, aber es gibt natürlich auch eine Papier-

Version für 44,99 €, z.B. bei Amazon: The Java ™ Tutorial: A Short Course on the Basics (Java Series)

- **Java ist auch eine Insel**

  Wird häufig empfohlen, aber ich persönlich mag den Stil nicht sonderlich. Vielleicht ist er mir *zu* praxisorientiert: Das Buch beschreibt zwar alles und jedes, bleibt mir aber oft zu sehr an der Oberfläche. Eher etwas zum schnellen Nachschlagen, wen man wissen will, wie man ein bestimmtes Problem in Java lösen kann. Unter http://openbook.galileocomputing.de/javainsel/ gibt es eine kostenlose Online-Version. Das gebundene Buch kostet 49,90 € und ist z.B. bei Amazon erhältlich: Java ist auch eine Insel

Weiterführendes und Vertiefendes

Die im Folgenden genannten Quellen sind besonders für alle diejenigen interessant, die sich über den Kurs hinaus praktisch mit Java beschäftigen wollen oder müssen. Teilweise werden Themen behandelt, die im Kurs keine Rolle spielen, in der Praxis aber durchaus wichtig sind, teilweise werden im Kurs behandelte Themen vertieft und man erfährt genauer, wie bestimmte Konzepte in Java realisiert sind und warum.

- **Java Programming Language Enhancements in JDK 5**

  Ein Überblick über die Spracherweiterungen, die mit der Version 5.0 zur Sprache Java hinzugekommen sind. Trotz der Kompaktjheit der Darstellung lesenswert: Generics, Foreach-Form der For-Schleife, Autoboxing/Unboxing, Enums, Varargs, Static Imports, Annotations... Nur online verfügbar auf http://download.oracle.com/javase/1,5.0/docs/guide/language/index.html

- **Die Kolumnen und Artikel von Angelika Langer**

  Schlichtweg unverzichtbar, meine absolute Empfehlung für jeden der mehr wissen will! Frau Langer behandelt verständlich und ausführlich so ziemlich jedes nichttriviale Thema, das für die Java-Praxis von Interesse ist. Wer wissen will, wie Generics (parametrische Polymorphie) in Java umgesetzt wurden, warum das so gemacht wurde und wo die Fallstricke lauern: Hier steht es. Wer wissen will, wie die Garbage Collection funktioniert, wie Enums unter der Haube arbeiten oder was man bei der Synchronisierung in Multithreading-Anwendungen beachten muss: Frau Langer erklärt es. Normalerweise neige ich wirklich nicht zu Groupie-haften Begeisterungsausbrüchen, aber die Frau ist einfach gut! In Anbetracht der Fülle der (teils auf Deutsch, teils auf Englisch) behandelten Themen nenne ich einfach ein paar Einstiegs-Links. Von dort aus einfach weiterstöbern...

  Eine Übersicht über alle Java-Artikel
  Java Generics FAQs - Frequently Asked Questions
  Die Artikel der Kolumne „EFFECTIVE JAVA“ aus der Zeitschrift „Java-Spektrum“
  Secrets of equals()