FAQ + Zusammenfassungen aus der Diskussions-Newsgroup zum Kurs 1618

Im Laufe der letzten Jahre sind im Rahmen der Betreuung der Newsgroup des Kurses 1618 eine ganze Menge Erläuterungen und Antworten auf immer wiederkehrende Fragen geschrieben worden. Einige davon haben wir überarbeitet und hier zusammengestellt. Die Erklärungen gehen dabei teilweise etwas über den Kurstext hinaus.

Letzte Bearbeitung des Dokuments: 18. Jun. 2021

Inhaltsverzeichnis

1 Fehlermeldungen beim Compilieren....4
1.1 This method must return a result of type... 4
1.2 Cannot make a static reference to the non-static method / field... 4
1.3 Cannot reduce the visibility of the inherited method... 4
1.4 Fehlermeldung bzw. Warnung im Zusammenhang mit @Override 5
1.5 Local variable X defined in an enclosing scope must be final or effectively final 6
2 Fehlermeldungen beim Ausführen....7
2.1 NullPointerException (NPE) 7
2.2 ArrayIndexOutOfBoundsException (AIOOBE) 8
2.3 ClassNotFoundException (CNFE) 8
2.4 IllegalMonitorStateException (IMSE) 8
3 Arrays....10
3.1 Grundlagen 10
3.2 Ich bekomme eine ArrayIndexOutOfBoundsException. 10
3.3 Ich bekomme beim Zugriff auf ein Array-Element eine NullPointerException. 10
3.4 Ich verwende eine Kopie eines Arrays, aber wenn ich darin Änderungen vornehme, wirken sich diese auf das Original aus. Warum? 11
3.5 Aber auch wenn ich den Array explizit kopiere, wirken sich Änderungen auf das Original aus! 11
3.6 Kann ich mit System.arraycopy() mehrdimensionale Arrays kopieren? 12
4 For-Each-Form der For-Schleife....13
4.1 Wie funktioniert die ForEach-Form der For-Schleife prinzipiell? 13
4.2 Kann ich in einer ForEach-Schleife keine Zuweisung an das aktuelle Element vornehmen? 14
4.3 Ich bekomme beim Iterieren über ein Array mit der Foreach-Schleife eine ArrayIndexOutOfBoundsException. 14
5 Call by value / Call by reference....15
5.1 Werden Parameter in Java "by value" oder "by reference" übergeben? 15
6 Casts....16
6.1 Wie kann ich den Typ X auf Y casten? 16
6.2 Der Compiler lässt mich eine Methode nicht aufrufen! 17
7 Das Schlüsselwort "this"....19
7.1 Grundlegende Bedeutung 19
7.2 Kann "this" jemals null sein? 20
7.3 Und in einer statischen Methode? 20
8 Verwendung von "static"....21
8.1 Wann darf/soll ich "static" verwenden? 21
8.2 Aber der Compiler verlangt doch nach "static"! 22
8.3 Gibt es denn auch sinnvolle Anwendungen von "static"? 23
8.4 Und wenn ich einfach nur Code aus einer Klasse verwenden will? 24
8.5 Wird static in Java nicht auch für Aufzählungstypen verwendet? 26
9 Statische vs. nichtstatische innere Klassen....27
9.1 Was ist der Unterschied? 27
9.2 Wann verwende ich was? 27

9.3 Es gibt also auch Instanzen statischer Klassen? 27
10 Aufzählungstypen in Java (Enum)......28
10.1 Ist ein Enum eine Klasse? 28
10.2 Kann ich über eine Enum iterieren, z.B. mit der ForEach-Form der For-Schleife? 28
10.3 Wo finde ich die API-Doku zu den Enum-Methoden values() und valueOf()? 28
10.4 In manchen Beispielen werden Enums innerhalb einer anderen Klasse definiert. Soll das so sein? 28
11 Super s = new Sub() – Warum?......29
12 Wann verwendet man "implements", wann "extends"?......30
13 Auflösung von Überladung, Überschreiben, dynamisches Binden......31
13.1 Was ist eigentlich dieses "Binden"? 31
13.2 Wie läuft das "Binden" bei Java ab? Was tut der Compiler und was die VM? 31
13.3 Wie funktioniert die Auflösung der Überladung durch den Compiler genau? 31
13.4 Und was tut dann die VM noch zur Laufzeit? 32
13.5 Warum ist es falsch, zu sagen, im zweiten Schritt des Most-Specific-Algorithmus würde die "am besten zum Aufruf passende" Methode ausgewählt? 32
13.6 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? 33
13.7 Warum wird im folgenden Beispiel nicht die Methode ausgeführt, die "3" ausgibt? Schließlich passt diese doch am besten zum Aufruf? 34
13.8 Aber warum ist das so? Warum wird nicht zur Laufzeit noch einmal nachgesehen, ob im dynamischen Typ des Empfängers eine besser passende Methode existiert? 35
13.9 Und die dynamischen Typen der übergebenen Parameter? Spielen die bei der Methodenwahl nie eine Rolle? 36
13.10 Bisher war bzgl. des "Bindens" nur die Rede von Methoden. Man sagt aber auch, in Java würden "Attributzugriffe statisch gebunden". Was heißt das? 36
13.11 Und *warum* werden Attributzugriffe statisch gebunden? Das ist doch verwirrend! 37
14 Kovarianz / Kontravarianz......38
15 Parallelität (Threads)......42
15.1 Wer führt eigentlich welchen Code in einem Java-Programm aus? (Eisenbahnbeispiel 1) 42
15.2 Wie funktioniert "synchronized"? (Eisenbahnbeispiel 2) 44
15.3 "Schützt" Synchronisation das Objekt, auf dem synchronisiert wird vor parallelen Zugriffen? 45
15.4 Wie erklärt das Eisenbahnbeispiel, dass ein Thread einen Monitor auch mehrfach sperren kann? 45
15.5 Und was ist mit wait/notify? 46
15.6 Fehlt noch notifyAll… 47

1 Fehlermeldungen beim Compilieren

1.1 This method must return a result of type...

Eine Methode, die nicht als void deklariert ist, muss sich, wenn sie sich nicht abrupt beendet, einen Wert des deklarierten Rückgabetyps zurückliefern. Wenn der Compiler dies nicht sicherstellen kann, tritt die o.g. Fehlermeldung auf.

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();
    }
}
```

Andere häufige Fälle sind If-Else-Kaskaden, bei denen nicht in jedem möglichen Fall ein Wert zurückgegeben wird und Switch-Anweisungen ohne Default-Fall.

1.2 Cannot make a static reference to the non-static method / field...

Statische Methoden sind Klassenmethoden, d.h. sie sind ihrer Klasse selbst zugeordnet und existieren unabhängig von Instanzen der Klasse. Deshalb gibt es in einer solchen Methode auch kein "This-Objekt", auf dessen Instanzvariablen oder Instanzmethoden man zugreifen könnte. Wenn Sie eine explizite Referenz auf ein Instanz haben, ist der Zugriff auf dessen Instanzvariablen oder Instanzmethoden natürlich möglich. Siehe dazu auch den Abschnitt "Verwendung von static".

1.3 Cannot reduce the visibility of the inherited method...

Eine Methode, die eine in einem Supertyp deklarierte Methode überschreibt/implementiert, darf keine geringere Sichtbarkeit haben, als die entsprechende Supertypmethode, da sonst eine Instanz der Subklasse nicht überall dort eingesetzt werden könnte, wo eine Instanz der Superklasse vorgesehen ist.

Dieser Fehler tritt gerne beim Implementieren von Interface-Methoden in einer Klasse auf: Interface-Methoden haben immer die Sichtbarkeitsstufe "public", ohne dass dies durch einen Sichtbarkeitsmodifikator deklariert werden muss. In einer Klasse deklarierte Methoden ohne Sichtbarkeitsmodifikator haben hingegen die Sichtbarkeitsstufe "package".

1.4 Fehlermeldung bzw. Warnung im Zusammenhang mit @Override

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 ("The method m() of type Foo must override or implement a supertype method").

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, der im Vorkurs verlinkt ist, wurde 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 eine entsprechende Warnung ("The method m() of type Foo should be tagged with @Override since it actually overrides a superinterface / superclass method").

1.5 Local variable X defined in an enclosing scope must be final or effectively final

Diese Meldung erhalten Sie, wenn Sie aus einer lokalen Klasse auf eine lokale Variable des Blocks zuzugreifen, in dem die lokale Klasse definiert ist. Das liegt daran, dass der Zugriff aus einer lokalen Klasse auf Variablen des umgebenden Blocks nur erlaubt ist, wenn diese als final deklariert sind (bis incl. Java 7) oder zumindest effektiv final sind (seit Java 8), ihnen also auf jeden Fall nach ihrer Initialisierung nichts mehr zugewiesen wird. Warum diese Einschränkung existiert, ist nicht ganz einfach zu verstehen und geht über den Kursstoff hinaus.

Da die Frage aber immer wieder aufkommt, hier der Versuch einer Erklärung:

Die Java-VM ist eine sog. Stackmachine. Das bedeutet vereinfacht, dass jeder Methodenaufruf seinen eigenen Block auf einem Stapel bekommt, in welchem er seine lokalen Variablen hält und mit dem er arbeiten kann. Kehrt eine Methode zurück, kann dieser sog. Stack-Frame entsorgt werden. Das heißt, dass alle methodenlokalen Variablen nur bis zum Ende ihrer Methode existieren.

Normalerweise ist das kein Problem, weil man von außen ohnehin nicht auf die lokalen Variablen einer Methode zugreifen kann. Wenn nun aber ein Objekt einer lokalen inneren Klasse (ab hier LIC abgekürzt) erzeugt wird, wird dieses im Normalfall länger leben, als die Methode, in der es deklariert wurde. Es könnte also passieren, dass man aus einer Methode des LIC-Objekts auf eine lokale Variable der "umgebenden Methode" zugreift, die zu diesem Zeitpunkt schon gar nicht mehr existiert.

Dieses Problem kann man lösen (und hat man gelöst), indem man schon bei der Erzeugung des Objekts der LIC dort eine lokale Kopie von der betroffenen Variablen anlegt. Damit kann deren Wert das Ende der Methode der umgebenden Klasse überleben.

Dies bringt nun allerdings ein weiteres Problem mit sich: Im weiteren Verlauf der "äußeren" Methode könnte sich nach der Erzeugung eines Objekts der LIC der Wert einer solchen Variablen ändern, so dass sich der Wert der zuvor angelegten Kopie (in der Instanz der LIC) ein anderer wäre als der aktuelle (oder nach dem Beenden der Methode als das letzte gültige) Wert. Um die draus resultierenden Zweideutigkeiten zu vermeiden, hat man festgelegt, dass von innerhalb einer LIC aus nur auf solche Variablen der umgebenden Methode zugegriffen werden kann, deren Wert nach ihrer Initialisierung nicht mehr verändert werden kann. Bis incl. Java 7 mussten solche Variablen als "final" deklariert sein, seit Java 8 reicht es, dass sie "effektiv final" sind, ihr Wert also nach ihrer Initialisierung nicht mehr geändert (was der Compiler für lokale Variablen ja leicht prüfen kann).

2 Fehlermeldungen beim Ausführen

2.1 NullPointerException (NPE)

Eine NPE tritt genau dann auf, wenn man versucht, das "Objekt hinter einer Referenz" anzusprechen, also einen Ausdruck 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 zu 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 ein Array, kann natürlich auch den Wert null haben.

Wenn man eine NPE bekommt, ist der erste Schritt also, herauszufinden, welcher Ausdruck, den 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();
```

so 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.

Typische Ursachen für NPEs sind z.B.:

- 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 Array, dessen Elementtyp ein Referenztyp ist, wurde zwar erzeugt, es wurden seinen "Slots" aber keine Referenzen auf Objekte zugewiesen. Damit haben die "Slots" des Arrays den Wert, den sie (analog zu Instanzvariablen anderer Objekte) bei der Objekterzeugung per automatischer Initialisierung bekommen haben, also den Wert null.

2.2 ArrayIndexOutOfBoundsException (AIOOBE)

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 zu dieser Form der For-Schleife finden Sie im entsprechenden Abschnitt dieses Dokuments.

2.3 ClassNotFoundException (CNFE)

Eine CNFE bedeutet, dass die Laufzeitumgebung eine Klasse nicht laden konnte, welche sie zur Ausführung des Programms benötigte. Wenn Sie Java gemäß der Anleitung im "Vorkurs" einrichten, sollte Ihnen die CNFE, wenn überhaupt, nur im Zusammenhang mit RMI (Kurseinheit 7) begegnen können.

Sollten Sie dort eine CNFE bekommen, dann haben Sie vermutlich die RMI-Registry nicht, wie im Kurs beschrieben, aus dem Serverprogramm heraus gestartet, sondern separat. Das ist zwar prinzipiell durchaus möglich, wird aber im Kurs nicht weiter behandelt.

2.4 IllegalMonitorStateException (IMSE)

Eine IMSE tritt genau dann auf, wenn ein Thread wait(), notify() oder notifyAll() auf einem Objekt aufruft, dessen Monitor dieser Thread nicht gesperrt hat. Der Aufruf der besagten Methoden richtet sich ja immer an ein ganz bestimmtes Objekt und bezieht sich auf den "Wait-Pool" dieses Objekts.

Im einfachsten Fall ist die Erklärung für eine IMSE also, dass man eine der drei Methoden aus einem Codebereich aufruft, der überhaupt nicht synchronisiert ist. Das

kann nicht funktionieren, denn dann hat der ausführende Thread ja überhaupt keinen Monitor gesperrt.

Dann gibt es den Fall, dass man eine der drei Methoden zwar schon aus 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 hier das von der Variablen referenzierte Objekt nach der ersten Zuweisung immer dasselbe sein *muss*.

3 Arrays

3.1 Grundlagen

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

- Array-Indexe zählen ab 0.
- Arrays sind Objekte. Ihre "Slots" verhalten sich in vielerlei Hinsicht wie die Instanzvariablen anderer Objekte.
- Die Elemente eines Arrays, dessen Elementtyp ein Referenz-Typ ist, sind Referenzen, nicht Objekte.
- Mehrdimensionale Arrays gibt es in Java genau genommen nicht. Das, was so aussieht, sind in Wirklichkeit eindimensionale Arrays, deren Elemente Referenzen auf weitere Arrays sind.

3.2 Ich bekomme eine ArrayIndexOutOfBoundsException.

Bitte lesen Sie hierzu den Unterabschnitt zur ArrayIndexOutOfBoundsException im Abschnitt "Fehlermeldungen beim Ausführen" in diesem Dokument.

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

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 das Array, nicht jedoch irgendwelche vordefinierten Vogel-Objekte. 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. 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 Defaultwert des entsprechenden Typs und bei Referenztypen ist dies nun einmal die leere Referenz null.

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

Evtl. haben Sie die vermeintliche Kopie etwa so erzeugt:

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

Damit erzeugen Sie aber gar keinen neuen Array, sondern Sie deklarieren lediglich eine neue Variable "kopie", welche nach der Zuweisung 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.

3.5 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 ungefähr 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 Methode System.arraycopy() macht also letztlich nichts anderes als das, was man auch mit einer For-Schleife machen könnte: Sie kopiert Werte um. Und da die Werte in diesem Fall Referenzen auf Objekte sind, werden auch nur Referenzen kopiert... die dann natürlich dieselben Objekte referenzieren wie ihre Originale. Dasselbe gilt übrigens auch bei Verwendung von Methoden wie Arrays.copyOf() oder clone().

3.6 Kann ich mit System.arraycopy() mehrdimensionale Arrays kopieren?

Ja, aber damit erreichen Sie nicht das, was Sie vermutlich wollen. Denn ein "mehrdimensionales Array" ist in Java in Wirklichkeit 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. Dasselbe gilt übrigens auch bei Verwendung von Methoden wie Arrays.copyOf() oder clone(). 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() o.Ä. einsetzen.

4 For-Each-Form der For-Schleife

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

Mit dieser Form der Schleife kann man über Arrays und über Subtypen des Interfaces Iterable (typischerweise sind das Collections, also z.B. Listen) iterieren, ohne explizit eine Indexvariable oder einen Iterator zu verwenden. Man gibt stattdessen Namen und Typ einer lokalen Variablen an, in die dann automatisch bei jedem Durchlaufen des Schleifenrumpfs der jeweils nächste Wert aus dem Arrays bzw. dem 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 Variable, in die
        // beim Iterieren nacheinander die Werte kopiert werden 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);
        }
    }
}
```

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

Sie können der Variablen, in der beim Iterieren nacheinander die Werte landen, zwar etwas zuweisen, aber diese Zuweisung wird sich nicht auf das Array oder Iterable auswirken, über das Sie iterieren. Diese Variable 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" in diesem Dokument). Eine Zuweisung an die Variable verändert also nicht das Iterable, über das iteriert wird. Wenn die Werte des Arrays oder Iterable Objektreferenzen sind, kann aber selbstverständlich über die in der Variablen landenden Kopien dieser Referenzen das referenzierte Objekt (durch Methodenaufruf oder Attributzuweisung) angesprochen und z.B. dessen Zustand verändert werden.

4.3 Ich bekomme beim Iterieren über ein Array mit der Foreach-Schleife eine ArrayIndexOutOfBoundsException.

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 i enthält hier also nicht, wie bei der klassischen Form, die Indexzahlen der Array-Slots, sondern vielmehr (eins nach dem anderen) Kopien der in den Slots enthaltenen Werte. Offenbar ist in Ihrem Fall eines dieser Elemente größer als der größte erlaubte Index des Arrays.

Eine korrekte Iteration mit der ForEach-Form der For-Schleife sähe also so aus (ich nenne die temporäre Variable hier bewusst nicht i, sondern elem, um anzudeuten, dass es sich eben nicht um eine Indexzahl handelt):

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

5 Call by value / Call by reference

5.1 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 Werte der Basisdatentypen "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 evtl. 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. Eine Variable enthält nie ein Objekt, sondern immer nur einen Wert und das ist entweder ein Wert eines der Basisdatentypen oder eine Referenz auf ein Objekt oder die "leere Referenz" null. Auch Methoden liefern keine Objekte, sondern ggf. Referenzen auf Objekte. Und wenn 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 dann 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 einer Methode keine Wirkung nach außen erzeugt: Der formale Parameter (der sich ja wie eine lokale Variable verhält) referenziert danach schlicht ein anderes Objekt, als der Ausdruck, der als aktueller Parameter verwendet wurde. 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!

Wenn Sie bei diesem Thema noch verunsichert sind, empfehle ich die Lektüre von https://javaranch.com/campfire/StoryCups.jsp und dann https://javaranch.com/campfire/StoryPassBy.jsp.

6 Casts

6.1 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 Basisdatentypen 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 persons[i] zur Laufzeit einen Studenten referenzieren wird, kann der Compiler nicht wissen. Wir können es ihm durch einen Cast zusichern. Der Aufruf sähe dann so aus:

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

Mit dem Cast 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 auf dem gecasteten Ausdruck. Eine Änderung des Typs eines Objekts erfolgt dabei nicht und kann auch nicht erfolgen: Ein Objekt behält immer für sein ganzes Leben den 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. Daran ändert auch der Cast nichts. Der Ausdruck (Student) p hingegen hingegen hat definitionsgemäß den Deklarationstyp Student, weswegen der Compiler die Zuweisung an die Variable s zulässt.

Ob der Cast dann auch zur Laufzeit funktioniert, hängt davon ab, ob das "gecastete Objekt" tatsächlich die ganze Zeit schon den Typ hatte, auf den gecastet wurde, was in unserem Beispiel der Fall ist: Das von p und s referenzierte Objekt *ist* ja ein Student.

In Fällen, bei denen sich zur Laufzeit herausstellt, dass das "gecastete Objekt" *nicht* von dem Typ ist, auf den gecastet wird (der Compiler also vom Programmierer "belogen" wurde), kommt es zu einem Laufzeitfehler (ClassCastException).

6.2 Der Compiler lässt mich eine Methode nicht aufrufen!

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");
    }
}
```

Hier meldet der Compiler einen Fehler bzgl. der Anweisung `sup.m()`, weil versucht wird, auf einem Ausdruck vom Deklarationstyp Super eine Methode aufzurufen, die für diesen Typ gar 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 m() aufgerufen wird zu Sub und der Compiler erlaubt nun den Aufruf. Man kann sich das auch so vorstellen, dass man dem Compiler durch den Cast zusichert, dass sup zur Laufzeit ein Sub-Objekt referenzieren wird. Eine solche Zusicherung hebelt natürlich letztlich die Typprüfung durch den Compiler aus: Sollte sup zur Laufzeit dann doch kein Sub referenzieren, kommt es zu einer ClassCastException.

7 Das Schlüsselwort "this"

7.1 Grundlegende Bedeutung

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 dieser Methode führte (auch als "impliziter Parameter" des Methodenaufrufs 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:

- Eine Instanzvariable ist durch eine gleichnamige lokale Variable oder einen formalen Parameter verdeckt, man will aber die Instanzvariable ansprechen. Generell sollte man derlei Verdeckungen am Besten vermeiden, es gibt aber einen typischen Fall, der völlig akzeptabel ist, nämlich eine Initialisierung von Instanzvariablen im Konstruktor durch übergebene Parameter. Beispiel:

  ```
  public class Person {
      private String name;

      Person(String name) {
          this.name = name;
      }
  }
  ```

  Hier ist das "this" nötig, damit klar ist, dass name bzw. auf der linken Seite der Zuweisung die Instanzvariable meint und nicht den gleichnamigen formalen Parameter.

- Aufruf eines anderen Konstruktors der gleichen Klasse. Beispiel:

  ```
  public class Person {
      private String name;

      Person(String name) {
          this.name = name;
      }

      Person() {
          this("Mustermann");
      }
  }
  ```

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

- 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. Z.B. könnte ein Hotelgast-Objekt sich selbst bei einem Weckdienst anmelden, damit dieser es später per Methodenaufruf wecken kann:

```
weckdienst.anmelden(this, "07:00");
```

- Aus einer (nichtstatischen) inneren Klasse heraus kann man das umgebende Objekt der äußeren Klasse normalerweise "direkt" ansprechen um auf dessen Instanzvariablen oder -methoden zuzugreifen. Wenn nun die innere Klasse selbst eine gleichnamige Instanzvariable oder -methode deklariert, dann verdeckt diese die des umgebenden Objekts. Um trotzdem auf die Elemente des umgebenden Objekts zugreifen zu können, existiert eine spezielle Syntax: Man verwendet den Klassennamen der äußeren Klasse, gefolgt von einem Punkt und dem Schlüsselwort "this". Beispiel:

```
public class Foo {
    private String name = "Aussen";

    class Bar {
        private String name = "Innen";

        void m() {
            System.out.println(name);
            System.out.println(Foo.this.name);
        }
    }
}
```

  Hier würde der Aufruf der Methode m() einer Instanz der Klasse Bar zuerst "Innen" ausgeben, dann "Aussen".

7.2 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 initialisiert 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.

7.3 Und in einer statischen Methode?

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.

8 Verwendung von "static"

8.1 Wann darf/soll ich "static" verwenden?

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). Instanzen der betreffenden Klasse sind dazu nicht notwendig.

Leider erlaubt Java es, statische Elemente *scheinbar* über Instanzen der betreffenden Klasse anzusprechen, also z.B. statische Methoden scheinbar auf Instanzen aufzurufen. In Wirklichkeit werden solche Aufrufe aber vom Compiler in Aufrufe über den Klassennamen umgewandelt. Aus dem Aufruf der statischen Methode m() im folgenden Code, der scheinbar auf einem Objekt erfolgt...

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

class Super {
    static void m() {
        System.out.println("Super");
    }
}

class Sub extends Super {
    static void m() {
        System.out.println("Sub");
    }
}
```

macht der Compiler einen Aufruf auf dem Klassennamen, der dem Deklarationstyp der Variablen s entspricht, also

```
        Super.m();
```

Man sollte sich unbedingt abgewöhnen, statische Zugriffe "über Instanzen" durchzuführen, weil es vernebelt, was man da eigentlich tut und damit das Verständnis erschwert. Schließlich wird der Aufruf einer statischen Methode statisch gebunden, d.h. es wird die Methode m der *Klasse* Super ausgeführt, nicht die des Objekts, welches von der Variablen s referenziert wird.

8.2 Aber der Compiler verlangt doch nach "static"!

Eine Ursache für den Anfängerfehler im Zusammenhang mit "static" liegt in der Tat 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. Oft verändert es seinen Zustand, welcher durch seine Instanzvariablen 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 unserer primitiven Simulation der Aufruf der Methode starte() auf einem Motorrad-Objekt, 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 der Klasse Motorrad, d.h. alle Motorräder 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 Objekt aufgerufen (hat keinen impliziten Parameter), sie *kennt* überhaupt keine Instanzen ihrer Klasse.

Kurz gesagt: Wenn ich ein Motorrad starten will, brauche ich auch ein Motorrad, also eine Instanz der Klasse Motorrad. 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.

8.3 Gibt es denn auch sinnvolle Anwendungen von "static"?

Ja, natürlich. Einige davon sind im Folgenden (ohne Anspruch auf Vollständigkeit) aufgeführt:

- Da wäre zum einen natürlich die main-Methode. Diese dient als Einstiegspunkt des Programms und ist static, denn sie ist nicht an Objekte gebunden. Dass sie sich - wie auch in meinem Motorradbeispiel - manchmal in einer Klasse befindet, von der man auch Instanzen 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.
- Eine weitere sinnvolle Verwendung von "static" sind die Methoden reiner Dienstleistungsklassen. Solche Klassen 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 Instanzen 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.
- Im Zusammenhang mit Factories findet man oft statische Methoden, so genannte Factory-Methoden. Eine solche Methode wird dann statt eines Konstruktors verwendet, wenn man eine Instanz eines bestimmten Typs benötigt. Einer der Vorteile ggü. einem Konstruktor besteht darin, dass die Factory-Methode - abhängig von irgendwelchen Faktoren - auch eine Instanz eines anderen Typs liefern als den der Klasse, zu der sie gehört. Oft ist das dann eine Instanz einer Subklasse dieser Klasse. Ein Beispiel wäre die Methode

getInstance() der abstrakten Klasse Calendar, welche abhängig von der verwendeten Locale und Zeitzone vorkonfigurierte Instanzen einer Subklasse von Calendar liefert. Auch im Kurs finden Sie ein entsprechendes Beispiel (ToyFactory in Kapitel 2.1).

8.4 Und 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 Dienstleistungsmethode 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. So etwas 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 {
   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 {
    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. Das könnte die Klasse YamahaZentralLager etwa so aussehen (bitte nur als Grobskizze verstehen):

```
public class YamahaZentralLager {
    private static int motorVorrat;

    public static void liefereMotor(Besteller b) {
        if (motorVorrat < 1) throw new NoMotorException();
        motorVorrat--;
        spedition.liefereAus(einMotor, b);
    }
```

```
}
```

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 für das Beispiel irrelevante Details wie etwa das Baujahr weggelassen habe und stattdessen den Besteller übergebe.

Schön ist das aber nicht. 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.

Besser wäre es, auch dieses eine Zentrallager bereits als "ordentliches Objekt" zu modellieren und sicherzustellen, dass man eine Referenz auf dieses eine Objekt bekommen kann, z.B. über eine (statische !) Factory-Methode. Wenn es dann später doch mehrere ZentralLager geben soll, kann man einfach diese Factory-Methode ändern, so dass sie z.B. je nach Standort des Bestellers eine Referenz auf das nächstgelegene ZentralLager liefert. Für den Besteller ändert sich damit aus seiner Sicht nichts, er muss also auch nicht angepasst werden. Und auch in der Klasse ZenralLager halten sich die Änderungen in Grenzen, weil Attribute wie etwa motorVorrat von Anfang an *Instanzvariablen* wären.

8.5 Wird static in Java nicht auch für Aufzählungstypen verwendet?

So hat man das in der Tat vor Java 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 5 glücklicherweise typsichere Aufzählungstypen (enum) gibt. Diese werden im Kurs in Kapitel 7 behandelt und auch hier in der FAQ gibt es ein Kapitel zu Enums. Eine ausführlichere (über den Kurs hinausgehende) Einführung mit Beispielen findet sich in https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html.

9 Statische vs. nichtstatische innere Klassen

9.1 Was ist der Unterschied?

Instanzen einer nichtstatischen inneren Klasse haben eine implizite Referenz auf eine bestimmte Instanz der umgebenden Klasse. Über diese implizite Referenz kann eine Instanz einer nichtstatischen inneren Klasse auf die Instanzvariablen und -methoden des umgebenden Objekts zugreifen, als wären es seine eigenen. Z.B. greift in Abb. 2.6 im Kurs ein ListIterator auf die Instanzvariable header "seiner" Liste zu. Eine Instanz einer nichtstatischen inneren Klasse kann ohne ein umgebendes Objekt nicht existieren.

Instanzen einer statischen inneren Klasse haben eine solche implizite Referenz nicht. Sie sind also "technisch" keiner bestimmten Instanz der umgebenden Klasse zugeordnet (auch wenn sie es "logisch" trotzdem sein können). Das bedeutet, dass Instanzen statischer innerer Klassen auch ohne ein umgebendes Objekt existieren können.

9.2 Wann verwende ich was?

Eigentlich ganz einfach: Wenn jede Instanz einer inneren Klasse immer zu einer ganz bestimmten Instanz der umgebenden Klasse gehört *und* Zugriff auf dessen Instanzvariablen oder -methoden benötigt, darf die innere Klasse nicht statisch sein. In allen anderen Fällen ist eine statische innere Klasse die bessere Wahl.

Beachten Sie aber bitte den "wichtigen Hinweis" am Ende von Abschnitt 2.2.2.1 im Kurs!

Absolut typische Fälle sind die beiden im Kurs gezeigten inneren Klassen Entry und ListInterator: Ein Entry benötigt keinen Zugriff auf Instanzvariablen oder -methoden der Liste, zu der er logisch gehört, ein Iterator benötigt diesen Zugriff hingegen zwingend, um seinen Job zu tun (über die Elemente der Liste zu iterieren).

9.3 Es gibt also auch Instanzen statischer Klassen?

Ja, natürlich. Eine statische innere Klasse ist, abgesehen von Kapselungsaspekten, im Wesentlichen eine ganz normale Klasse, von der auch Instanzen existieren können und typischerweise auch existieren. Siehe etwa die statische innere Klasse Entry in der Klasse LinkedList1618 in Abb. 2.4 im Kurs.

Wenn Sie glaubten, dass es keine Instanzen statischer Klassen geben kann, haben Sie diese vielleicht mit Utility-Klassen wie java.lang.Math verwechselt, die lediglich dazu dienen, diverse Funktionen und Konstanten von überall her über den Klassennamen ansprechbar zur Verfügung zu stellen. Von solchen Klassen können in der Tat meist keine Instanzen erstellt werden. Erreicht wird dies, indem explizit ein privater Konstruktor deklariert wird. Mit statischen Klassen hat das aber nichts zu tun!

10 Aufzählungstypen in Java (Enum)

10.1 Ist ein Enum eine Klasse?

Bei "enum" handelt es sich zunächst einmal um ein Schlüsselwort, das dazu dient, eine Typdefinition einzuleiten, so wie "class" und "interface". 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. 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.

10.2 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 Instanzen der Enum-Klasse liefert. Und über dieses Array kann man dann ganz normal iterieren, auch mit der ForEach-Schleife.

10.3 Wo finde ich die API-Doku zu den Enum-Methoden values() und valueOf()?

Leider überhaupt nicht. Es handelt sich um Methoden, die weder im Quellcode Ihrer Enum-Klasse noch in dem der Klasse Enum existieren, weswegen das Javadoc-Werkzeug, mit dem die API-Doku aus dem Quellcode erstellt wird, für diese Methoden auch keine Einträge erzeugen kann. Diese beiden (statischen) Methoden werden vielmehr vom Compiler automatisch für jede Enum-Klasse erstellt und in den Bytecode geschrieben. Man muss also als Java-Programmierer wissen, dass es sie gibt und wie sie einzusetzen sind.

10.4 In manchen Beispielen werden Enums innerhalb einer anderen Klasse definiert. Soll das so sein?

Nein, ganz im Gegenteil! In der Praxis wird man eine Enum so gut wie nie als innere Klasse modellieren wollen. Denn Enum-Typen sind normalerweise sehr "zentrale" Typen eines Programms: Man findet sie oft als Parameter oder Rückgabetypen von Methoden, über die verschiedene funktionale Untereinheiten eines Programms miteinander kommunizieren. Es wäre also ziemlich unsinnig, sie innerhalb einer anderen Klasse zu "verstecken". Dass man dies in Code-Beispielen trotzdem manchmal findet, liegt nur daran, dass der jeweilige Autor sein Beispiel möglichst kompakt halten wollte.

11 Super s = new Sub() – Warum?

In Java-Quellcode findet man oft folgendes Muster:

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

D.h. es wird eine Instanz einer Klasse Sub erzeugt und eine Referenz auf diese Instanz wird in einer Variablen abgelegt, deren Deklarationstyp aber nicht Sub ist, sondern ein Supertyp von Sub, wobei es sich bei dem Supertyp sowohl um eine Klasse als auch um ein Interface handeln kann. Beispiele:

```
Component comp = new Label("Text");

List myList = new ArrayList();
```

Für diese Vorgehensweise gibt es zwei wichtige Gründe:

1. Durch die Verwendung des Supertyps als Deklarationstyp der Variablen wird für den Leser des Codes automatisch dokumentiert, dass auf der Variablen nur Aufrufe solcher Methoden erfolgen werden, die im Supertyp auch deklariert sind.
2. Durch die Verwendung des Supertyps muss bei einer späteren Änderung des verwendeten Subtyps nur diese eine Stelle angepasst werden. Z.B. könnte in einem Programm, in dem die Variable myList wie oben als vom Typ List deklariert wurde, die ArrayList später ohne Weiteres gegen eine LinkedList ausgetauscht werden.

Es ist deswegen sinnvoll und üblich, für Variablendeklarationen möglichst weit oben in der Typhierarchie stehende Typen zu verwenden. "Möglichst weit oben" ist dabei natürlich nicht absolut zu verstehen (sonst würde man ja überall Object nehmen), sondern so, dass der "höchste" Typ verwendet wird, der die Methodenschnittstelle anbietet, die bei der nachfolgenden Verwendung der Variablen benötigt wird.

12 Wann verwendet man "implements", wann "extends"?

Kurze Antwort:

Vergessen Sie bitte alle Auswendiglern-Regeln der Form "Wenn ein Interface beteiligt ist, verwendet man implements", denn diese Regeln sind *falsch*. Sie verwenden "implements" nur in genau *einem* Fall, nämlich dann, wenn Sie in einer Klassendeklaration festlegen wollen, dass die Klasse ein Interface implementiert. Für *alle* anderen Fälle, in denen irgendeine Subtypbeziehung postuliert werden soll, verwenden Sie "extends".

Ausführlicher:

Generell gilt, dass ein Subtyp eine "Erweiterung" (engl. extension) seiner Supertypen darstellt. Der Begriff "Erweiterung" kommt hier daher, dass ein Subtyp typischerweise mehr Eigenschaften hat und/oder mehr Nachrichten versteht als seine Supertypen. "Erweitern" ist also der allgemeine Begriff für das, was ein Subtyp ggü. seinem Supertyp tut. Man *hätte* also schlicht das "extends" für jede Form der Subtypbeziehung nehmen können.

Aber da explizit als solche benannte Interfaces bei der Einführung von Java ein neues und besonderes Sprachfeature waren, wollte man die Art der Subtypbeziehung, bei der ein Klassentyp Subtyp eines Interfacetyps ist (dieses also "implementiert") besonders hervorheben. Und deshalb hat man genau für diese eine Art von Subtypbeziehung das Schlüsselwort "implements" vorgesehen.

Für alle anderen Variationen, in denen eine Subtypbeziehung postuliert werden soll, also für

- Subclassing (Klassentyp erweitert Klassentyp)
- Subtyping zwischen Interfaces (Interfacetyp erweitert Interfacetyp)
- Obere Schranken (Typparameter muss Subtyp des Schrankentyps sein)

verwendet man "extends".

13 Auflösung von Überladung, Überschreiben, dynamisches Binden

13.1 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.

13.2 Wie läuft das "Binden" bei Java ab? Was tut 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 dem Aufruf eine ganz bestimmte Methodensignatur zugeordnet und ein entsprechender Aufruf wird in den Bytecode geschrieben, oder aber es kommt zu einem Compilerfehler, nämlich dann, wenn der Compiler unter mehreren zum Aufruf passenden Methodensignaturen keine Entscheidung treffen kann.

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

13.3 Wie funktioniert die Auflösung der Überladung durch den Compiler genau?

Vorbemerkung: Bitte beachten Sie im Folgenden, dass die Subtypbeziehung definitionsgemäß reflexiv ist, d.h. jeder Typ ist Subtyp (und Supertyp) seiner selbst!

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 Methodensignaturen angelegt, das sind alle, für die gilt, dass die Deklarationstypen der aktuellen Parameter des Aufrufs Subtypen der Deklarationstypen der formalen Parameter der betreffenden Methodensignatur sind. Besteht diese Liste aus nur einem Element, wird direkt der entsprechende Methodenaufruf in den Bytecode geschrieben.

Wenn die Liste der zum Aufruf passenden Methodensignaturen hingegen mehrere Elemente enthält, ist ein weiterer Schritt erforderlich, die Auswahl der speziellsten Methodensignatur. Dazu wird aus der Liste jede Methodensignatur gestrichen, für die es in der Liste eine speziellere gibt. Der Vergleich, welche von zwei

Methodensignaturen spezieller ist, ist dabei völlig unabhängig von einem konkreten Methodenaufruf: Eine Methodensignatur M2 ist genau dann spezieller als eine Methodensignatur 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 Methodensignatur übrig, ist diese die speziellste der zum Aufruf passenden Methodensignaturen und ein entsprechender Aufruf wird in den Bytecode geschrieben. Bleiben mehrere Methodensignatur übrig, dann sind diese offenbar gleich speziell, und es ist nicht möglich, den Methodenaufruf eindeutig aufzulösen. → Compilerfehler.

13.4 Und was tut dann die VM noch zur Laufzeit?

Als Ergebnis der Tätigkeit des Compilers steht im Bytecode ein konkreter Aufruf einer Methode mit einer ganz bestimmten Signatur. Zur Laufzeit muss nun die Laufzeitumgebung ("Virtual Machine", VM) diesem Aufruf 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.

Anders gesagt: Dynamisches Binden ist ein technisches Mittel, um eins der wesentlichen Grundprinzipien des objektorientierten Paradigmas umzusetzen: "Das Objekt entscheidet, wie es auf eine Nachricht reagiert."

Da in Java eine Methode nur dann eine gleichnamige Methode eines Supertyps überschreibt, wenn die Methodensignaturen (also Methodenname, Anzahl und 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 lassen muss, ist klar, sonst hätte der Compiler den entsprechenden Methodenaufruf ja gar nicht erst zugelassen.

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

Weil aus der Liste der prinzipiell zum Aufruf passenden Methodensignaturen die speziellste Methodensignatur ausgewählt wird. Dazu wird aus der Liste jede Methodensignatur gestrichen, zu der es eine speziellere gibt. Die Frage, welche von zwei Methodensignaturen spezieller ist, ist aber völlig unabhängig vom einem konkreten Methodenaufruf: Eine Methodensignatur M2 ist genau dann spezieller als

eine Methodensignatur 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 Methodensignaturen ("passendste Methodensignatur") führt leicht zu dem Fehler, dass auch Methodensignaturen einbezogen werden, die im Deklarationstyp des Ausdrucks, auf dem der Aufruf erfolgt, gar nicht vorhanden sind (weil sie nur in Subtypen dieses Typs existieren).

13.6 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) { … }
```

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. Ein Zählen von Abständen wäre auch nicht sinnvoll, denn:

1. Eine "logische Ebene" einer Typhierarchie ist Ausdruck einer Abstraktionsstufe in Bezug auf die abstrakte Sicht auf die Realwelt, die in der Typhierarchie modelliert wird. Diese Gliederung dieser Abstraktion kann aber bei zwei Parametern, die ja eine unterschiedliche Bedeutung haben, auch unterschiedlich fein/grob ausfallen, so dass die eine Typhierarchie vielleicht insgesamt nur drei Stufen, die andere aber fünf hat.

2. Zusätzlich zu den in 1. beschriebenen "logischen Ebenen" findet man in einer Typhierarchie oft noch etwas, was ich hier mal als "technische Ebene" bezeichnen möchte: Ebenen, die in erster Linie aus implementierungstechnischen Gründen eingefügt wurde, z.B. abstrakte Klassen, in denen gemeinsame Implementierungsteile der darunter liegenden Ebenen zusammengefasst wurden.

Fazit: 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 SUB→ 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.

13.7 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 { }
```

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 eine Sub-Instanz 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 für diese auf den ersten Blick willkürlich erscheinende Beschränktheit des Compiler ist, dass es für den Compiler (der ja nur einzelne Klassen

compiliert) gar nicht möglich ist, lokal zu analysieren, was einer Variable im Laufe ihrer Existenz so zugewiesen wird. So könnte eine Variable z.B. mit dem Resultat des Aufrufs einer dynamisch gebundenen Methode einer Instanz einer anderen Klasse belegt werden. Die Variable könnte auch eine Instanzvariable eines Objekts sein, auf dem mehrere Threads operieren können. Damit besteht die Möglichkeit, dass die Threads den Wert der Variablen in nicht vorhersagbarer Reihenfolge ändern, womit es ebenfalls unmöglich wird, vorherzusagen, welches Objekt sie zu einem bestimmten Zeitpunkt referenzieren wird.

Es spielen also für den Compiler bei Methodenaufrufen auf der Variablen sup nur die Methodensignaturen eine Rolle, welche für deren Deklarationstyp Super deklariert sind (in diesem selbst oder von einem seiner Supertypen geerbt). Der Compiler wählt unter diesen Signaturen diejenigen aus, welche zum Methodenaufruf passen. Das sind genau die, für die gilt, dass die Deklarationstypen aller Aufrufparameter Subtypen der entsprechenden formalen Parameter in der Methodendeklaration sind. In unserem Beispiel passt nur die Methode "1" zum Aufruf. In den Bytecode wird also ein Aufruf einer Methode mit der Signatur 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-Objekt referenziert, beginnt die VM mit ihrer Suche nach der Methodensignatur, deren Aufruf im Bytecode steht, in der Klasse Sub. Würde also in Sub eine Methode existieren, welche die Methode m(Tier, Tier) aus Super überschreiben, 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 (unterschiedliche Signatur).

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.

13.8 Aber warum ist das so? Warum wird nicht zur Laufzeit noch einmal nachgesehen, ob im dynamischen Typ des Empfängers eine besser passende Methode existiert?

Eine Methode ist ja das technische Mittel, mit dem ein Objekt eine ganz bestimmte Nachricht verarbeitet. Identifiziert werden Methoden über ihre Signatur. Eine Methodensignatur besteht aus dem Bezeichner der Methode sowie Anzahl und Typen der formalen Parameter. Wenn jemand in einer Subklasse eine Methode überschreibt, dann will er, dass eine ganz bestimmte Nachricht auf eine andere Weise erfolgt, als das in der überschriebenen Methode der Superklasse der Fall ist. Eine überschreibende Methode muss daher die gleiche Signatur haben wie die überschriebene, denn sie soll ja für die Verarbeitung derselben Nachricht stehen!

Nehmen wir nun an, es gibt im dynamischen Typ des Nachrichtenempfängers eine "besser zum Aufruf passende" Methode als die, die der Compiler gewählt hat. Dann

hat diese Methode ja offenbar eine Signatur, die im Deklarationstyp des Ausdrucks, auf dem der Aufruf erfolgte, gar nicht vorhanden war, sonst hätte der Compiler ja diese Methodensignatur ausgewählt. Es handelt sich also um eine *andere* Methode, zwar mit gleichem Bezeichner, aber unterschiedlichen formalen Parametern. Die Subtypmethode steht also für eine Nachricht, deren Verabeitung im Supertyp schlicht nicht vorgesehen war.

Und genauso, wie Sie, wenn Sie auf einem Supertyp-Ausdruck eine Methode foo() aufrufen, erwarten würden, dass das zur Laufzeit dazu führt, dass stattdessen eine im Subtyp vorhandene Methode bar() ausgeführt wird, sollten Sie erwarten, dass eine Methode mit einer anderen Parameterliste ausgeführt wird.

Überladene Methoden sind *verschiedene* Methoden, die für *verschiedene* Nachrichten stehen. Der Compiler wählt anhand des Aufrufs eine bestimmte Methodensignatur aus und es wird dann *diese* Methode des Empfängerobjekts ausgeführt, keine andere.

13.9 Und die dynamischen Typen der übergebenen Parameter? Spielen die bei der Methodenwahl nie eine Rolle?

Für Java: Nein, sie spielen keine Rolle, weder für den Compiler noch für die VM. Es gibt aber objektorientierte Programmiersprachen, bei denen die Methodenwahl nicht nur anhand des dynamischen Typs des Nachrichtenempfängers erfolgt, sondern auch anhand der dynamischen Typen der aktuellen Parameter. Man spricht dabei von sog. Multimethoden bzw. von "multiple dispatch".

13.10 Bisher war bzgl. des "Bindens" nur die Rede von Methoden. Man sagt aber auch, in Java würden "Attributzugriffe statisch gebunden". Was heißt das?

Richtig, der Begriff "Binden" wird auch bei Attributen verwendet. Hier meint er – analog zur Begriffsverwendung bei Methoden – den Vorgang, bei dem einer Attributselektion im Quellcode das Attribut 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. Und da Attributzugriffe in Java statisch gebunden werden (also bei der Übersetzung der Klasse Test durch den Compiler), wird der Zugriff anhand des Deklarationstyps der Variablen a "gebunden", also an die Variable a, die in Super deklariert ist.

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").

13.11 Und *warum* werden Attributzugriffe statisch gebunden? Das ist doch verwirrend!

Wie so oft, wenn beim Entwurf einer Programmiersprache seltsam anmutende Entscheidungen getroffen werden, geht es auch hier um Performance: Statisches Binden findet durch den Compiler statt, bei dynamischem Binden muss die VM zur Laufzeit noch Entscheidungen treffen, was natürlich Zeit kostet.

In solchen Fällen trifft man eine Abwägung. Und die ist bei Java für das statische Binden von Attributen ausgefallen. Das ist aber letztlich auch weniger dramatisch, als man auf den ersten Blick meinen könnte.

Wenn man sich an die Regel hält, direkte Zugriffe auf Instanzvariablen von Objekten einer anderen Klasse möglichst zu vermeiden und solche Zugriffe über Getter- und Setter-Methoden abwickelt (die ja dynamisch gebunden werden), löst sich die Verwirrung in Wohlgefallen auf und alles verhält sich wieder so, wie man es eigentlich erwarten würde.

Darüber hinaus kommt die Verwirrung ohnehin nur zum Zuge, wenn man in einer Super- und einer Subklasse gleichnamige Instanzvariablen hat. Und dafür gibt es schlicht keinen vernünftigen Grund. Der einzige denkbare Grund wäre, dass man in der Subklasse die Superklassenvariable überschreiben will. Aber warum sollte man eine Instanzvariable überschreiben können? Man überschreibt ja Methoden, um das Verhalten einer Methode im Subtyp anzupassen. Aber Instanzvariablen *haben* ja so etwas wie ein Verhalten überhaupt nicht.

14 Kovarianz / Kontravarianz

Wichtiger Hinweis vorab: Die folgenden Erörterungen beziehen sich *nicht* auf irgendeine bestimmte Programmiersprache, sondern es geht zunächst darum, was unter bestimmten Voraussetzungen *logisch* möglich wäre und was nicht. Nicht alles, was logisch möglich ist, ist auch sinnvoll und nicht alles, was logisch möglich wäre, ist in jeder OO-Sprache auch umgesetzt. Das betrifft insbesondere auch Java: Was in Java bzgl. Ko- und Kontravarianz erlaubt und verboten ist, wird erst am Endes des Kapitels behandelt. Trotzdem habe ich mich aus Gründen der Lesbarkeit bzgl. der Notation an Java orientiert.

Die konzeptionelle Grundlage, von der wir in diesem Kapitel ausgehen, ist die Definition der Subtypbeziehung: Jedes Objekt eines Typs T ist auch Objekt aller Supertypen von T und darf deshalb überall dort stehen, wo ein Objekt eines der Supertypen erlaubt ist.

Die Frage, bei der die Begriffe Ko-/Kontravarianz eine Rolle spielen, ist nun 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.

In unserem Kontext 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 Parametertyps 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.

Aus der o.g. Definition der Subtypbeziehung ergibt sich, dass ein Subtyp-Objekt die Fähigkeiten des Supertyp-Objekts, 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 oder
- einen Rückgabewert liefert, den nicht auch die Supertyp-Methode hätte liefern können oder
- sich abrupt mit einer Ausnahme beendet, mit der sich nicht auch die Supertyp-Methode hätte beenden 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.

Schauen wir uns diese beiden Aspekte einmal an einem Beispiel an. Gegeben seien folgende Subtypbeziehungen:

Kuh **SUB**→ Tier **SUB**→ Object   Sub **SUB**→ Super

Anders gesagt: Jede Kuh **ist** ein Tier, jedes Tier **ist** ein Object, jedes SUB **ist** ein Super.

1. Wenn es nun in der Klasse Super eine Methode m() gibt...

   Klasse Super `Tier m() { ... }`

   und die Klasse Sub diese Methode überschreiben will, was ist dann rein logisch zulässig?

   a) Klasse Sub `Kuh m() { ... }` (kovarianter Rückgabetyp)

   b) Klasse Sub `Tier m() { ... }` (novarianter Rückgabetyp)

   c) Klasse Sub `Object m() { ... }` (kontravarianter Rückgabetyp)

   Antwort: Da eine Nachricht, die an einen Ausdruck vom Typ Super gesendet wird, aufgrund von Polymorphie und dynamischer Methodenwahl jederzeit bei einem Objekt vom Typ Sub ankommen kann, darf die überschreibende Methode nichts zurückliefern, was die überschriebene nicht auch hätte liefern können. Logisch möglich sind also die Varianten a) und b). Überschreiben mit kontravariantem Rückgabetyp ist logisch nicht zulässig, denn

   ```
   Super s = new Sub();
   Tier t = s.m();
   ```

   würde zu einem Typfehler führen, da die Variable t die gelieferte Referenz auf ein Object nicht aufnehmen kann.

2. Wenn es nun in der Klasse Super eine Methode m() gibt...

   Klasse Super `void m( Tier t ) { ... }`

   und die Klasse Sub diese Methode überschreiben will, was ist dann rein logisch zulässig?

   a) Klasse Sub `void m( Kuh k ) { ... }` (kovarianter Parametertyp)

   b) Klasse Sub `void m( Tier t ) { ... }` (novarianter Parametertyp)

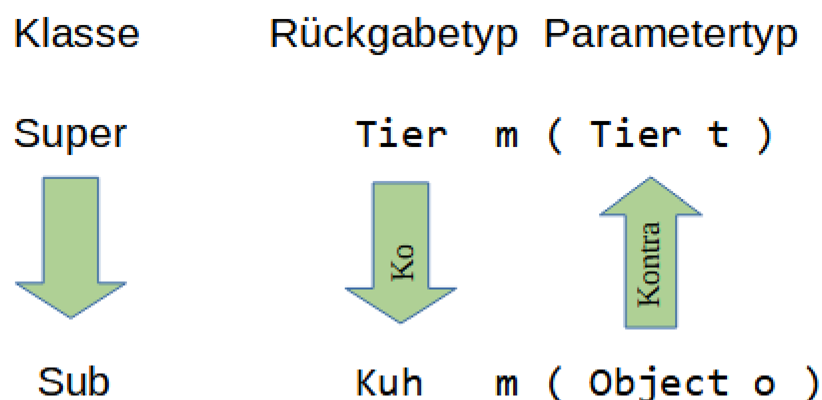
   c) Klasse Sub `void m( Object o ) { ... }` (kontravarianter Parametertyp)

   Antwort: Da eine Nachricht, die an einen Ausdruck vom Typ Super gesendet wird, aufgrund von Polymorphie und dynamischer Methodenwahl jederzeit bei einem Objekt vom Typ Sub ankommen kann, muss die überschreibende Methode mindestens die Parameter "verkraften", die die überschriebene verkraftet. Logisch möglich sind also die Varianten b) und c). Überschreiben mit kovariantem Parametertyp ist logisch nicht zulässig, denn

   ```
   Super s = new Sub();
   s.m(new Tier());
   ```

   würde zu einem Typfehler führen, da die Methode m() eines Sub das übergebene Tier nicht als Parameter "verkraftet".

Die logisch möglichen Beziehungen beim Überschreiben lassen sich also folgendermaßen veranschaulichen, wobei der Pfeil jeweils den "Schritt" vom Supertyp zum Subtyp symbolisiert:



Wenn wir "vom Supertyp zum Subtyp gehen", müssen wir bzgl. einer überschreibenden Methode beim Rückgabetyp "in die gleiche Richtung gehen" (deshalb Kovarianz), bei den Parametertypen "in die entgegengesetzte Richtung gehen" (deshalb Kontravarianz).

Zusätzlich ist bzgl. Rückgabetyp und Parametertypen auch Novarianz logisch möglich, d.h. der Typ ist in der überschreibenden und der überschriebenen Methode derselbe.

"Logisch möglich" bedeutet hier: "logisch möglich, unter der Voraussetzung, dass die am Anfang getroffene Definition des Subtyping gilt ("Überall dort, wo ein Objekt vom Typ Super stehen kann, ist auch ein Objekt vom Typ Sub erlaubt"), und dass Aufrufe von m() dynamisch gebunden werden.

"Logisch möglich" heißt aber *nicht*, dass die jeweilige Möglichkeit in einer konkreten Programmiersprache auch so umgesetzt sein muss. So erlaubt z.B. erlaubt Java beim Überschreiben zwar Kovarianz bzgl. des Rückgabetyps, bzgl. der Parametertypen ist aber nur Novarianz erlaubt, d.h. eine Methode eines Subtyps überschreibt eine Methode eines Supertyps nur dann, wenn diese dieselbe Signatur hat.

!

Und: "Logisch möglich" bedeutet auch nicht unbedingt "sinnvoll": Überschreiben mit kontravariantem Parametertyp ist im obigen Sinne zwar logisch möglich, aber *nicht* sinnvoll: In der Realität haben speziellere Typen auch speziellere Methoden und zwar sowohl bzgl. des Rückgabetyps als auch bzgl. der Eingangsparameter!

Beispiel: Wenn Tiere "Nahrung" fressen, fressen spezielle Typen speziellere Nahrung, nicht allgemeinere. Insofern ist mit "Überschreiben mit kontravariantem Parametertyp" leider nichts Sinnvolles anzufangen. Es ist also kein Verlust, dass Java dies nicht erlaubt.

Will man Sachverhalte wie z.B. "Kühe sind spezielle Tiere, die eine spezielle Art Nahrung fressen, nämlich Gras" modellieren, kann man das in Java und ähnlichen Sprachen mit den Mitteln der parametrischen Polymorphie erreichen, wobei der Preis dafür eine Einschränkung der Subtypbeziehung ist. So ist in Java z.B. der Typ `Tier<Nahrung>` *nicht* Supertyp des Typs `Tier<Gras>`, auch wenn Gras Subtyp von Nahrung ist.

Darüber hinausgehend gibt es OO-Sprachen (z.B. Eiffel), die tatsächlich "Überschreiben mit kovarianten Parametertypen" erlauben. Das steht im Gegensatz zu all dem oben Gesagten und entsprechend ist dafür dann auch wieder ein Preis zu bezahlen, nämlich Einschränkungen bzgl. der Polymorphie bei den beteiligten Typen und/oder bzgl. des dynamischen Bindens von auf solche Weise überschriebenen Methoden.

15 Parallelität (Threads)

15.1 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) {
        TestThread t = new TestThread();
        t.start();
        t.dotry();
        t.stop();
    }
}

class TestThread 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 {
            System.in.read();
        } 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 VM-Thread und dem Objekt der Klasse Thread, welches ja lediglich eine objektorientierte Abstraktion des VM-Threads im Programm 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-Objekt ist ein ganz normales Objekt, mit Methoden wie jedes andere, also ein Teil des Schienennetzes. Dass einige der Methoden dazu führen, dass ein VM-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 ZM (für "Zug Main"). ZM fährt nun die main()-Methode ab und kommt zu folgender Zeile:

```
TestThread t = new TestThread();
```

Es wird also ein Objekt der Klasse TestThread erzeugt, die von der Klasse Thread abgeleitet ist. Damit haben wir erst einmal 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 VM-Thread. Nennen wir diesen Z2. Z2 wird automatisch auf den Beginn der run()-Methode des von t referenzierten Thread-Objekts gesetzt und fährt, sobald der Scheduler ihm Rechenzeit gibt, unabhängig von ZM 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 ZM? 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: ZM befährt die Methode dotry() des von t referenzierten Objekts:

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

Hier funktioniert die read()-Methode wie ein Stopp-Signal: ZM steht so lange, bis eine Zeile von der Konsole gelesen werden kann, also Return gedrückt wurde. Währenddessen befährt Z2 fröhlich die Schleife in der run-Methode des ihn repräsentierenden Thread-Objekts.

Wird nun Return gedrückt, fährt ZM wieder los (sobald er vom Scheduler Rechenzeit bekommt), es wird die Abschiedszeile ausgegeben, und ZM kehrt zu der Stelle zurück,

wo er in die Methode dotry() des Thread-Objekts abgebogen ist, und gelangt nun zu dieser Zeile:

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

Das heißt, ZM biegt in die stop()-Methode des von t referenzierten Objekts ein. Auch hier ist für ZM wieder egal, dass es sich bei dem Objekt um eines 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 VM-Thread, nämlich Z2. Und die Methode stop() des Objekts hat genau den Zweck, den repräsentierten Thread zu beenden. Genau das geschieht nun: *PLOPP*. Der Zug Z2 löst sich in Wohlgefallen auf. Das von t referenzierte Objekt allerdings existiert erst einmal noch fröhlich weiter, wie es ja auch schon existierte, als "sein" Thread Z2 noch gar nicht in der Welt war.

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

15.2 Wie funktioniert "synchronized"? (Eisenbahnbeispiel 2)

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 gleichzeitig einen so gesicherten Bereich befährt. Aber wir wollen mehr:

Wir wollen nämlich auch erreichen können, dass in bestimmten verschiedenen Streckenabschnitten immer nur insgesamt ein Zug unterwegs ist. Die Notwendigkeit hierfür ergibt sich daraus, dass die betreffenden Gleisabschnitte nicht wirklich unabhängig voneinander sind. Wir können uns z.B. vorstellen, dass es Kreuzungen gibt. D.h., wenn ein Zug in den Abschnitt A einfährt, muss nicht nur das Signal für Abschnitt A auf rot gehen, sondern auch das für einen weiteren Abschnitt B (und umgekehrt). Und erst, wenn in keinem der Abschnitte mehr ein Zug ist, sollen beide Signale wieder grün werden.

Das wird nun - weil zu komplex - nicht mehr über direkte Verschaltungen 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 für mehrere Streckenabschnitte die zugehörige Kontrollstation 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 oben postulierten Kreuzungspunkte verschiedener Gleise entsprechen Ressourcen, die in einen inkonsistenten Zustand geraten könnten, wenn mehrere Threads parallel auf sie zugreifen. Und 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 zwingend. 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 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, keine Basisdatentypen. Wenn man also konkurrierende Zugriffe auf eine Variable eines Basisdatentyps 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 Objekts synchronisiert werden soll, möglich sein, eine Referenz auf das Objekt zu erhalten.

15.3 "Schützt" Synchronisation das Objekt, auf dem synchronisiert wird vor parallelen Zugriffen?

Das *kann* so sein. Nämlich genau dann, wenn alle Methoden eines Objekts, die es erlauben, dessen Attribute zu ändern, als synchronized markiert sind und ein Zugriff auf Attribute auch ausschließlich über diese Methoden möglich ist. Sind diese Voraussetzungen aber nicht gegeben, z.B. weil auch direkte Zugriffe auf die Attribute möglich sind oder es zugreifende Methoden gibt, die nicht synchronisiert sind, kann es zu allen Problemen kommen, die mit konkurrierenden Zugriffen verbunden sein können.

15.4 Wie erklärt das Eisenbahnbeispiel, dass ein Thread einen Monitor auch mehrfach sperren kann?

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 auch nicht in einen anderen Bereich einfahren, der ebenfalls durch die Kontrollstation kontrolliert wird, die er selbst gesperrt hat, obwohl dieser Vorgang ja völlig unproblematisch wäre. D.h., wir müssen das Modell ergänzen:

Um die mögliche Mehrfachsperrung in das Eisenbahnmodell einzubauen, müssen wir zusätzlich postulieren, dass ein Rotsignal für genau einen Zug nicht gilt, nämlich für den, dessen Einfahrt die Rot-Schaltung überhaupt erst ausgelöst hat. Dieser Zug darf, nachdem er einmal in den von einer bestimmten Kontrollstation überwachten Bereich eingefahren ist, alle Rotsignale ignorieren, die zu eben dieser Kontrollstation gehören. Allerdings zählt die Kontrollstation mit, wie viele "ihrer" Gleisabschnitte dieser Zug insgesamt betreten hat. Und erst, wenn er genau so viele wieder verlassen hat schaltet die Kontrollstation wieder auf grün.

15.5 Und was ist mit wait/notify?

Wait/notify wird in Situationen eingesetzt, in denen ein Thread, um weiterarbeiten zu können, darauf angewiesen ist, dass ein anderer Thread durch seine Tätigkeit bestimmte Voraussetzungen herstellt. Auch das können wir auf das Eisenbahnbeispiel übertragen.

Stellen wir uns vor, wir haben es mit altertümlichen Dampflokomotiven zu tun. Einer unserer Züge – Z – fährt in ein Gleis ein, an dem sich eine Kohlendepot befindet, um dort Kohle aufzunehmen. Dazu hat Z natürlich die Kontrollstation dieses Gleises auf rot gestellt. Nun stellt Z aber fest, dass das Depot leer ist. Damit Z nun Kohle aufnehmen und dann weiterfahren könnte, müsste ein anderer Zug neue Kohle zum Lager bringen. Nach unserem bisherigen Modell müsste dazu Z erst wieder wegfahren und die Kontrollstation zum Zufahrtsgleis freigeben. Dann könnte Z später einen neuen Versuch starten… so lange, bis irgendwann Kohle vorhanden ist. Wie man sich vorstellen kann, wäre das allerdings ziemlich ineffizient („busy waiting“).

Deshalb ergänzen wir unser Modell nun: Wenn Z feststellt, dass keine Kohle vorhanden ist, kann er der Kontrollstation, die er auf rot geschaltet hat, die spezielle Nachricht wait() senden. Das führt dazu, dass Z in einen speziellen Wartebereich verschoben wird, über den jede Kontrollstation verfügt und in dem mehrere Züge Platz haben. Gleichzeitig wird durch dieses Vorgang die Kontrollstation wieder auf grün geschaltet. Damit kann nun ein Versorgungszug V das Depot anfahren (wobei er die Kontrollstation auf rot schaltet) und mit Kohle befüllen.

Um nun unseren Zug Z, der sich im Wartebereich befindet, wieder ins normale

Gleissystem zu bringen, kann ein Zug, der eine Kontrollstation auf rot geschaltet hat – in unserem Fall wäre das V, dieser die Nachricht notify() senden. Natürlich darf dadurch Z nicht direkt wieder auf das Versorgungsgleis fahren. Denn dort befindet sich ja noch V. Deswegen wird Z zwar fahrbereit gemacht, der Weg vom Wartebereich zurück ins normale Gleissystem ist aber ebenfalls mit einem Signal der betreffenden Kontrollstation versehen. Somit kann Z das Depot erst anfahren, wenn V durch Verlassen des Versorgungsgleises die Kontrollstation freigegeben hat. Dabei muss Z natürlich die Kontrollstation wieder auf rot schalten, damit kein anderer Zug in das Versorgungsgleis einfahren kann. Wenn das nicht geht, weil ein dritter Zug ihm zuvorgekommen ist, bleibt Z ganz normal vor dem Signal stehen, bis die Kontrollstation wieder grün ist.

15.6 Fehlt noch notifyAll...

Der einzige Unterschied ggü. notify() ist, dass bei notifyAll() nicht nur *einer* der Züge im Wartebereich einer Kontrollstation wieder fahrbereit wird, sondern *alle*. Das wäre in unserem Szenario durchaus sinnvoll, schließlich wird V ja nicht so wenig Kohle in das Depot gebracht haben, dass es nur für einen Zug reicht. Natürlich kann trotzdem immer nur einer der Züge gleichzeitig zum Depot fahren, aber das wird ja, wie oben beschrieben, durch das zusätzliche Signal zwischen Wartebereich und Kontrollstation sichergestellt. In diesem Szenario ist natürlich besonders wichtig, dass die aus dem Wartebereich ausfahrenden Züge am Depot noch einmal prüfen, ob dort immer noch Kohle vorhanden ist und sich anderenfalls erneut per wait() in den Wartebereich begeben.