Hinweise zu den Codebeispielen:

Die Java-Codebeispiele sind in einem Eclipse-Projekt namens „Studientag_1618“ enthalten. Dieses Projekt liegt als Zip-Datei vor, welche Sie in Ihr Workspace-Verzeichnis entpacken können. Anschließend legen Sie dann das Projekt aus Eclipse heraus mit dem o.g. Namen neu an. Wer kein Eclipse hat, findet den Quellcode im ausgepackten Projektverzeichnis unter src.

Projekt Studientag_1618

Package bsp_01 _deklarationstyp

1. Subyppolymorphie in ihrer primitivsten Form:

Die Zeile 5 zeigt, dass man einer Referenz-Variablen nicht nur eine Referenz zuweisen kann, die vom selben Typ ist wie die Variable, sondern auch Referenzen die als Deklarationstyp einen Subtyp des Deklarationstyps der Variablen haben. Im Beispiel steht rechts des Zuweisungsoperators das Ergebnis eines Konstruktoraufrufs. Ein solcher Ausdruck hat ja bekanntlich als deklarierten Rückgabetyp immer den Typ der Klasse, von der er ein Exemplar erzeugt.

2. Der Compiler interessiert sich nur für Deklarationstypen.

Der Compilerfehler in Zeile 6 entsteht, weil versucht wird, auf einem Ausdruck vom Deklarationstyp Super eine Methode aufzurufen, die für diesen Typ nicht definiert ist („The method m() is undefined for the type Super“).

Package bsp_02 _casting

Hier geht es um Casts. Bekanntlich ist ein Cast eine Zusicherung an den Compiler, dass das von einer Referenz referenzierte Objekt zur Laufzeit einen ganz bestimmten Typ haben wird. So eine Zusicherung kann vom Compiler naturgemäß so gut wie nicht überprüft werden, weswegen jeder notwendige Cast auch mit dem Verlust der Typsicherheit einhergeht.

In Zeile 7 haben wir zunächst einmal keinen Cast, sondern den bekannten Fall, dass der Compiler uns auf einem Ausdruck nur Methodenaufrufe / Attributzugriffe erlaubt, die für den Deklarationstyp des Aufrufs auch definiert sind. Und für Object – der Deklarationstyp von o1 – ist nun einmal kein Attribut name deklariert.

In Zeile 9 haben wir nun einen Cast, d.h. wir sichern dem Compiler zu, dass o1 zur Laufzeit ein Person-Exemplar referenzieren wird. Das glaubt er uns. Denn von dem, was in Zeile 6 auf der rechten Seite steht, weiß er nichts, er kennt nur den Deklarationstyp von o1, und eine Variable vom Typ Object kann ja zur Laufzeit tatsächlich ein Person-Exemplar enthalten, den Person ist Subtyp von Object. Und da der Compiler den Cast akzeptiert, erlaubt er uns auch den Attributzugriff in Zeile 10, denn eine Person hat ein Attribut name.

Interessant ist nun die Frage, warum der Compiler für den fast genauso gelagerten Fall in Zeile 13 den Cast *nicht* akzeptiert. Nun, der Deklarationstyp von s ist String. Und daher kann s auch zur Laufzeit niemals eine Person referenzieren. Denn dazu müsste Person Subtyp von String sein. Nun weiß der Compiler zwar i.A. gerade *nicht*, welche Subtypen es von einem Typ irgendwann einmal geben könnte. Aber im Fall von String weiß er es, denn die Klasse String ist als final deklariert, und das heißt, dass von ihr keine Subtypen abgeleitet werden *können*.

Warum akzeptiert der Compiler aber den Cast in Zeile 16? Schließlich sichern wir ihm zu, dass p3 zur Laufzeit ein Objekt referenzieren wird, welches das Interface Printable implementiert. Die Variable p3 ist

aber vom Deklarationstyp Person und die Klasse Person implementiert Printable nicht, was ja auch der Compiler sehen kann. Richtig… aber p3 könnte zur Laufzeit ein Exemplar eines Subtyps von Person referenzieren. Und zwar ein Exemplar eines Subtyps, der sehr wohl Printable implementiert! Wir könnten schließlich eine Klasse Student schreiben, mit „Student extends Person implements Printable“. Und ob das der Fall sein wird, kann der Compiler nicht entscheiden, weswegen er genötigt ist, uns zu glauben.

Dass wir gelogen haben, kann erst zur Laufzeit erkannt werden. Wenn wir die Zeilen 6 – 13 auskommentieren, also nur die Zeilen 15 – 17 in main() bleiben, können wir das Programm kompilieren und ausführen. Es kommt zu einer ClassCastException in Zeile 16.

Kommen wir zu Zeile 19. Auf der rechten Seite der Zuweisung haben wir einen Konstruktoraufruf. Dessen Deklarationstyp ist definitionsgemäß der Typ der Klasse, deren Konstruktor aufgerufen wird, also Object. Der Compiler akzeptiert den Cast auf Person, denn er ist prinzipiell möglich: Eine Referenz vo Deklarationstyp Object *könnte* ja auch eine Person referenzieren. Dass das hier mit Sicherheit nicht der Fall ist, kann der Compiler nicht feststellen, denn er vergleicht ja nur Deklarationstypen.

Zur Laufzeit würde (wenn wir den Rest so auskommentieren, dass keine Compilerfehler mehr auftreten und das Programm bis zu dieser Stelle läuft) auch Zeile 19 zu einer ClassCastException führen. Denn die VM stellt fest, dass das erzeugte Objekt eben doch keine Person *ist*. Hier haben wir übrigens den interessanten Fall, dass auch ein Cast, der gar nicht nötig gewesen wäre, zu einem Laufzeitfehler führen kann. Ohne den Cast wäre Zeile 19 ja völlig unproblematisch, sowohl für den Compiler als auch zur Laufzeit. Gute IDEs warnen einen vor solchen unnötigen Casts. Solche Warnungen weisen zwar nicht auf Compilerfehler hin (sonst wären sie nicht nur Warnungen), aber sie sind meist ein Anzeichen für logische Fehler oder Missverständnisse des Programmierers.

Package bsp_03_dynamische_bindung

Sowohl sup1 als auch sup2 sind vom Deklarationstyp Super. Der Compiler schreibt also einen Aufruf der Methode m() aus Super in den Bytecode. Zur Laufzeit stellt die Laufzeitumgebung fest, dass sup1 tatsächlich ein Super referenziert, sup2 hingegen ein Sub. Durch dynamische Methodenwahl sorgt die Laufzeitumgebung dafür, dass genau die Methode m() des Nachrichtenempfängers ausgeführt wird. Da im Typ Sub die Methode m() überschrieben wurde, ist die Ausgabe also:

```
Methode m von Super
Methode m von Sub
```

Package bsp_04_statische_bindung_bei_static

Fast das Gleiche wie im vorigen Beispiel, nur sind die Methoden m() diesmal static! Aber auch hier werden die Methoden m() *scheinbar* auf Objekten aufgerufen. In Wirklichkeit haben statische Methoden aber kein Empfängerobjekt, sie werden über den Klassennamen angesprochen. Leider erlaubt Java die irreführende Art des Aufrufs, die im Beispiel gezeigt wird. In Wirklichkeit wandelt aber der Compiler die Aufrufe in die Form um, die man für statische Elemente ausschließlich verwenden sollte, nämlich in Aufrufe über den Klassennamen.

Eclipse weist uns auf die unangebrachten Aufrufe mit mehreren Warnings hin: „The static method m() from the type Sub should be accessed in a static way“. Mit Hilfe der Quickfix-Funktion kann man Eclipse die Aufrufe umwandeln lassen. Das Ergebnis wäre für die Methode main() der Klasse Test:

```
Super sup1 = new Super();
Super.m();
Super sup2 = new Sub();
Super.m();
Sub.m();
```

wobei die Objekterzeugungen in der ersten und dritten Zeile natürlich völlig irrelevant sind. Das waren sie natürlich vorher auch schon, aber jetzt sieht man es.

Package bsp_05_statische_bindung_von_attributen

Attributzugriffe werden in Java statisch gebunden (also durch den Compiler), und zwar nicht nur Klassenattribute (static), sondern auch „normale“ Attribute. Deshalb ist bzgl. der Auflösung des Attributzugriffs in Zeile 6 auch nur der Deklarationstyp von s relevant und es wird das Attribut a der Klasse Super ausgegeben.

Zeile 7 zeigt einen ganz normalen dynamisch gebundenen Methodenaufruf: Zur Laufzeit wird die Methode m1() von Sub gewählt und da diese im Code der Klasse Sub definiert ist, *muss* sie auf das Attribut a von Sub zugreifen!

Zeile 8 zeigt, dass in der Klasse Sub das geerbte Attribut a aus Super zwar durch das gleichnamige Attribut von Sub *verdeckt* wurde, aber trotzdem nach wie vor vorhanden ist: Ein Sub hat ja auch die Methoden aus Super geerbt und kann sie aufrufen. Genau das tut die Methode m2() in Zeile 32 mit der geerbten Methode x(), Und da x() eine Methode ist, die im Code der Klasse Super definiert ist, kann *ihr* Attributzugriff sich nur auf das Attribut aus Super beziehen. Das in Zeile 8 erzeugt Sub-Exemplar hat also offenbar zwei Attribute a, von denen eines verdeckt/versteckt, aber über geerbte Methoden dennoch zugreifbar ist. Damit man nicht aus Versehen Attribute von Superklassen verdeckt, warnt uns Eclipse bzgl. der Deklaration in Zeile 25: „The field Sub.a is hiding a field from type Super.“

Package bsp_06_konstruktoren_1

Zeile 4: Die Klasse Lebewesen hat keinen explizit angegebenen Konstruktor. Dennoch wird hier ein Exemplar erzeugt. Erklärung: Hat eine Klasse keinen explizit deklarierten Konstruktor, bekommt sie automatisch einen parameterlosen Default-Konstruktor, dieser wird hier verwendet.

Zeile 5: Die Klasse Person *hat* einen explizit deklarierten Konstruktor. Dadurch entfällt der implizite Default-Konstruktor. →Compilerfehler „The constructor Person() is undefined“

Zeilen 22, 32, 40: In den Klassen Person, Student, Professor werden für das zu initialisierende Attribut die gleichen Bezeichner verwendet wie für den Parameter, der zur Initialisierung verwendet wird. Das ist bei Konstruktoren gängige Praxis. Ein Parameter ist jedoch bzgl. seiner Methode bzw. hier seines Konstruktors eine lokale Variable und verdeckt das gleichnamige Attribut. Durch Voranstellen von this wird jeweils klargestellt, dass auf der linken Seite der Zuweisung die Variable des aktuellen Objekts gemeint ist, also das Attribut.

Zeile 31: In der Klasse Student verwendet der Konstruktor in der ersten Zeile explizit einen Konstruktor der Superklasse.

Zeile 39: Jeder Konstruktor, der *nicht* explizit in der ersten Zeile einen anderen Konstruktor der gleichen Klasse oder einen Konstruktor der Superklasse aufruft, ruft implizit immer den parameterlosen Konstruktor der Superklasse auf. Da Person aber keinen parameterlosen Konstruktor hat, geht dies hier nicht und es kommt zu einem Compilerfehler „Implicit super constructor Person() is undefined. Must explicitly invoke another constructor“. Zur Beseitigung würde man analog zu Student vorgehen.

Package bsp_07_konstruktoren_2

Dieses Beispiel demonstriert ein Problem, das entstehen kann, wenn aus einem Konstruktor Methoden aufgerufen werden und eine Subklasse diese überschreiben darf. Erläuterung: Der Superklassenkonstruktor wird immer vor Rumpf des Subklassenkonstruktors ausgeführt. Der Superklassenkonstruktor

ruft die Methode m() auf. Diese wurde in der Subklasse überschrieben. Aufgrund dynamischer Bindung – das aktuelle Objekt ist das, welches gerade erzeugt wird, und das ist ja vom Typ Sub – wird die Methode m() der Subklasse gewählt. Diese will auf dem Subklassenattribut eine Methode aufrufen, aber das Subklassenattribut ist noch gar nicht initialisiert. → NullPointerException

Package bsp_08_abstrakte_klassen

Das Beispiel zeigt, wie man aus einer konkreten Methode einer abstrakten Klasse abstrakte Methoden der konkreten Subklassen aufrufen kann (Zeile 12), obwohl ja völlig unbekannt ist, wie deren Implementierung in diesen Subklassen aussehen wird. *Dass* eine solche Implementierung in jeder konkreten Subklasse vorhanden sein wird, wird eben durch die Deklaration der abstrakten Methode in der abstrakten Klasse (Zeile 17) erzwungen und vom Compiler überprüft.

Package bsp_09_polymorphie

Die Klassen dieses Packages zeigen die Verwendung und die Eigenschaften von Subtyppolymorphie, parametrischer Polymorphie und beschränkter parametrischer Polymorphie, wie im PDF „oop-konzepte.pdf“ im Abschnitt "Polymorphie" beschrieben. Um das Beispiel möglichst einfach zu halten wurde ein minimalen Behälter verwendet, der genau ein Element aufnehmen kann.

- Klasse A_SubtypPolymorphie: Es wird Object als Elementtyp verwendet, man kann also beliebige Objekte in den Container füllen, bekommt diese aber auch nur als Object zurück → Cast nötig, Verlust an Typsicherheit
- Klasse B_ParametrischePolymorphie: Löst das vorgenannte Problem: Der Verwender des Containers legt den Elementtyp bei dessen Erzeugung fest, die Notwendigkeit des Casts entfällt. Der Erbauer des Containers weiß nichts darüber, welchen Elementtyp der Verwender wählen wird.
- Klasse C1_BeschraenkteParametrischePolymorphie_Versuch1: Ein Versuch, innerhalb des Containers einen Vergleich anzustellen, scheitert, weil der Compiler keine Annahmen über den Elementtyp treffen kann und daher auf dem Element nur den Aufruf von Methoden erlaubt, die für Object definiert sind.
- Klasse C2_BeschraenkteParametrischePolymorphie_Versuch2: Der Programmierer des Containers hat als obere Schranke für den Typparameter den Typ Comparable angegeben. Damit kann der Verwender nur noch solche Elementtypen wählen, die Subtyp von Comparable sind. Vorteil: Der Compiler kennt die Schranke und kann daher innerhalb des Containers den Aufruf der Methode compareTo() auf dem Element erlauben. Das Beispiel ist in dieser Vereinfachung natürlich nicht wirklich sinnvoll. Ein komplexeres aber dafür praxisnahes Beispiel wäre ein binärer Suchbaum, bei dem Sie ja zur Einordung der Element diese ebenfalls vergleichen müssen. Bei der Klasse C2_BeschraenkteParametrischePolymorphie_Versuch2 erhalten wir zwei Warnungen, dass wir den Typ Comparable ohne Typparameter verwenden, obwohl dessen Deklaration einen Typparameter vorsieht. Das Problem, vor dem wir hier gewarnt werden, ist wieder ein Typproblem: Der Verwender könnte als Elementtyp einfach Comparable selbst nehmen, womit man auch Objekte verschiedener Klassen, die Comparable implementieren, in den Container füllen könnte. Typischerweise führt dies aber zu einer ClassCastException beim Vergleich mit compareTo().
- Klasse C3_BeschraenkteParametrischePolymorphie: Die Warnungen wurden beseitigt, indem bei der Angabe der Schranke für den Typparameter des Containers dieser erneut verwendet wurde. Die dahinterstehende Logik ist, dass wir als Schranke ja nicht einfach allgemeine Vergleichbarkeit haben wollen, sondern Vergleichbarkeit eben mit dem Elementtyp, den der Verwender wählt. Damit das funktioniert, muss der Elementtyp natürlich deklarieren, vergleichbar mit sich selbst zu sein. Genau das ist für die Typen, die Comparable implementieren, i.A. der Fall. So implementiert z.B. der Typ String nicht einfach Comparable, sondern eben Comparable<String>.

Package bsp_10_visibility

Hier geht es um die verschiedenen Modifikatoren, mit denen sich die Sichtbarkeit von Klassen, Methoden, Attributen usw. beeinflussen lassen. In jeder der vier Klassen findet sich ein Attribut vom Typ A, sowie eine Methode test(), in der versucht wird, auf dem Attribut seine vier Methoden aufzurufen. die jeweils nach ihrer Sichtbarkeitsstufe benannt sind. Bemerkenswert sind dabei vor allem folgende Sachverhalte:

- Aus einem Exemplar der Klasse A sind private Methoden eines *anderen* Exemplars dieser Klasse sichtbar. Die Kapselung durch Sichtbarkeitsmodifikatoren findet als offensichtlich nicht auf der Exemplar- sondern auf der Klassenebene statt.
- Aus einem Exemplar der Klasse B (im selben Package wie A) sind neben öffentlichen Methode und solchen mit Default-Sichtbarkeit auch die mit der Sichtbarkeit mit der Sichtbarkeit protected sichtbar. Wie man also sieht: Die Sichtbarkeitsstufe protected schließt die Default-Sichtbarkeit (gleiche Package) ein (das liest man immer wieder anders).
- Aus einem Exemplar der Klasse C sind nur die öffentlichen Methoden des A-Exemplars sichtbar. Das zeigt, dass das Package a.a *kein* Teil des Packages a ist, d.h. die Packages bilden in Java *keine* echte Hierarchie.
- Aus einem Exemplar der Klasse D (welche Subklasse von A ist) sind die als protected deklarierten Methoden eines A-Exemplars *nicht* sichtbar, wenn die Referenz, über die zugegriffen wird, den Deklarationstyp A hat. Hat sie hingegen den Deklarationstyp D, *sind* sie sichtbar, obwohl ja ein D-Exemplar auch ein A-Exemplar ist. Diese Spezialität der Sichtbarkeitsstufe protected ist teilweise auch langjährigen JavaProgrammierern nicht bekannt.

Package bsp_11_awt

Dieses Beispiel ist ein minimales AWT-Beispiel: Ein Fenster mit einem Panel und einem Button. Einige Aspekte sind hier erwähnenswert:

1. In der main()-Methode wird lediglich der Konstruktor aufgerufen, sonst passiert dort nichts. Dennoch ist das Programm nach der Ausführung von main() offenbar nicht zu Ende, sondern läuft weiter. Daraus kann man schließen, dass mit der Erzeugung und Sichtbarmachung des Fensters implizit ein weiterer Thread neben dem Main-Thread gestartet wird. Das ist tatsächlich der Fall, es handelt sich um eine ganze Gruppe von Threads, die sich um die Ereignisverarbeitung des AWT kümmern. Wenn das Fenster erzeugt ist, wartet der sogenannte Event Dispatch Thread (EDT) auf Ereignisse, die an dem Fenster oder seinen Komponenten auftreten. Diese werden in eine Ereigniswarteschlange eingereiht und dann nacheinander anbgearbeitet, d.h. es werden für jedes Ereignis die ereignisverarbeitenden Methoden der bei der Ereignisquelle für dieses Ereignis angemeldeten Listener ausgeführt.

2. In Zeile 20 erzeugen wir ein Panel-Exemplar. Zeile 21 zeigt, dass auch aus einem Konstruktor bereits Methoden des gerade in Erzeugung begriffenen Objekts aufgerufen werden können. Und in Zeile 22 wird das Panel mit Hilfe der von Frame geerbten Methode add() unserem MyFrame-Exemplar hinzugefügt. Ein Frame ist Subtyp von Container. Container sind all jene AWT-Komponenten, die als „Behälter“ für andere Komponenten dienen können. „Behälter“ ist hier bezogen auf die Oberfläche, also nicht im Sinne von Container-Klassen wie LinkedList oder ArrayList.

3. In Zeile 24 wird ein Button erzeugt, der in Zeile 30 ebenfalls mit add() dem Container als Oberflächenelement hinzugefügt wird. Diesmal wird add() allerdings nicht nur die Komponente (der Button) übergeben, sondern ein weiterer Wert. Offensichtlich ist die Methode add() der Klasse Frame überladen, d.h. sie existiert mehrmals mit unterschiedlichen Parametertypen. Bei dem zweiten Wert handelt es sich um ein statisches Attribut der Klasse BorderLayout, welches angibt, wo der Button erscheinen soll. Das

AWT verwendet sog. Layoutmanager, welche den Containern zugeordnet werden und für die Anordnung der Elemente des Containers sorgen. Ein Frame hat als Vorgabe ein BorderLayout und der übergebene Parameter sorgt dafür, dass der Button im Bereich SOUTH des Frames erscheinen wird. Das Panel hingegen wird in den Bereich CENTER gelegt werden: Wenn bei add() kein Bereich angegeben wird, legt ein BorderLayout die Komponente ins CENTER.

4. In den Zeilen 24-29 wird bei dem Button ein Listener angemeldet. Die Methode addActionListener() erwartet als Parameter ein Objekt, welche das Interface ActionListener implementiert. Statt nun eine eigene Klasse als Subtyp von ActionListener zu implementieren, wird hier eine sog. blocklokale anonyme innere Klasse verwendet: Wir erzeugen ein Exemplar einer Klasse, über welche nur bekannt ist, dass sie das Interface ActionListener implementiert, wobei die Methode actionPerformed() so implementiert wird, wie in den Zeilen 25-28 beschrieben. Die Klasse wird also ad hoc deklariert, ohne dass sie einen Klassennamen bekommt. Das gleiche Verfahren wird später für den WindowListener angewendet, wobei dort die anonyme Klasse von WindowAdapter (eine Implementierung von WindowListener mit leeren Methodenrümpfen) abgeleitet wird und die eine Methode überschreibt, für die wir uns im Listener tatsächlich interessieren.

Package bsp_12_exception

Klasse Test_1:

In Zeile 7 wird eine Methode aufgerufen, welche deklariert, dass sie eine OutOfBoundsException auslösen kann. Da eine OutOfBoundsException eine Checked Exception (Subklasse von Exception, aber nicht von RuntimeException) ist, muss der Methodenaufruf entweder in einen try/catch-Block mit passender Catch-Klausel gekapselt wird oder die Methode, in der der Aufruf steht, muss slebst deklarieren, die betreffende Exception auslösen zu können. Keines von beidem ist hier gegeben →Compilerfehler. Sehen Sie sich an, welche Quickfix-Optionen Eclipse bietet, um den Compilerfehler zu beseitigen.

Im darauffolgenden Codeblock wurde der Aufruf nun in try/catch gekapselt. Es gibt *mehrere* Catch-Blöcke, von denen beim Abfangen einer Exception immer der erste passende gewählt wird. Außerdem exisitert ein Finally-Block. Dieser wird unabhängig davon *immer* ausgeführt, ob der dazugehörige try-Block abrupt terminierte oder normal beendet wurde.

In der Methode aMethod() sehen wir, wie eine Exception ggf. tatsächlich ausgelöst wird, nämlich mit dem Schlüsselwort *throw*, gefolgt von einem Ausdruck, der ein Exception-Objekt referenziert, typischerwiese ist dies der Aufruf eines Konstruktors, der ein solches Objekt erzeugt.

Die Klasse OutOfBoundsException schließlich zeigt, dass eine Exception-Klasse eine ganz normale Klasse ist, deren Exemplare Attribute und Methoden haben können. Das Attribut value wird hier verwendet, um beim Auslösen der Exception die Information weiterzuleiten, *welcher* Wert eigentlich zur Auslösung geführt hat.

Klasse Test_2:

Hier haben wir in der Methode m1() einen ein catch-Block, der den Ausnahmetyp OutOfBoundsException behandeln soll. Die Klasse Out-OfBoundsException ist Subklasse von Exception, aber nicht von RuntimeException, es handelt sich also um eine Checked Exception. Da der Compiler erkennt, dass diese Exception im betreffenden try-Block nicht ausgelöst werden *kann* – im try-Block befindet sich ja kein Aufruf einer Methode, die deklariert, eine solche Exception auslösen zu können – meldet er einen Fehler " Unreachable catch block for OutOfBoundsException. This exception is never thrown from the try statement body" (Eclipse Compiler).

In der Methode m2() haben wir einen try-Block mit zwei catch-Blöcken, von denen der erste den Ausnahmetyp Exception, der zweite den Ausnahmetyp OutOfBoundsException behandeln soll. Da aber bei Auftreten einer OutOfBoundsException bereits der erste catch-Block "passt" (OutOfBoundsException ist ja Subtyp von Exception), kann der zweite catch-Block nie zur Ausführung kommen. Dies merkt der Compiler und meldet einen Fehler "Unreachable catch block for OutOfBoundsException. It is already handled by the catch block for Exception" (Eclipse Compiler).

Package bsp_13_enums

Die Enum Wochentag repräsentiert Wochentage, d.h. jedes Exemplar dieser Klasse steht für einen Wochentag. Wie man sieht, kann eine Enum neben der Aufzählung ihrer Werte auch Methoden enthalten. *Jede* Enum verfügt über die Methoden toString() und ordinal(). Letztere liefert die Stelle, an der der betreffende Enum-Wert in der Aufzählung steht (beginnend mit 0). Außerdem verfügt eine Enum-Klasse immer über die statischen Methoden values(), welche die für diese Enum-Klasse definierten Werte als Array liefert und valueOf(String name), welche zu einem String das entsprechend benannte Exemplar der Enum liefert, falls ein solches existiert. Ist für die Enum kein Wert definiert, der dem übergebenen String entspricht, wird eine IllegalArgumentException ausgelöst.

Sehen Sie sich zu diesem Beispiel und zum Thema "Enums" bitte auch den entsprechenden Abschnitt in der FAQ an!

Package bsp_14_threads

Das erste Beispiel (Konto1.java) zeigt Code, der in einer Multithreading-Umgebung zu Daten-Inkonsistenzen führen kann. Problemszenario: Thread 1 betritt die Methode einzahlen() und liest, um den rechten Teil der Zuweisung auszuwerten, den Wert von „saldo“ und addiert den „betrag“ hinzu. Nun unterbricht der Scheduler diesen Thread und Thread 2 wird rechnend. Auch dieser betritt die Methode einzahlen(), liest den saldo, addiert einen btrag auf und schreibt den neuen saldo zurück. Nun wird wieder Thread 1 rechnend und schreibt den neuen saldo zurück, den er vorhin errechnet hat. Ergebnis: Die von Thread 2 vorgenommene Einzahlung ist verloren gegangen.

Das zweite Beispiel (Konto2.java) zeigt Code, der in einer Multithreading-Umgebung zu einem Deadlock führen kann. Problemszenario: Thread 1 möchte einen Betrag von Konto A auf Konto B umbuchen, Thread 2 einen Betrag von Konto B auf Konto A. Thread 1 betritt die Methode umbuchenAuf() von Konto A und sperrt damit den Monitor von „this“, das ist hier Konto A. Nun wird Thread 2 rechnend und betritt die Methode umbuchenAuf() von Konto B und sperrt damit den Monitor von „this“, das ist hier Konto B. um weiterarbeiten zu können, müsste nun Thread 1 den auf „ziel“ synchronisierten Block betreten, also den Monitor von Konto B sperren, denn das ist ja für ihn „ziel“.. Diesen hat aber bereits Thread 2 gesperrt. Also kann Thread 1 erst weiterarbeiten, wenn Thread 2 diesen Monitor freigibt. Das wird aber nie geschehen, denn um weiterarbeiten zu können, müsste Thread 2 den auf „ziel“ synchronisierten Block betreten, also den Monitor von Konto A sperren, denn das ist für ihn „ziel“. Somit wartet jeder der beiden Threads auf den anderen → Deadlock.

Das dritte Beispiel (RunnableVsThread.java) zeigt die beiden Möglichkeiten, einen Thread in Java zu modellieren: Die Klasse Worker1 ist von Thread abgeleitet und überschreibt deren Methode run(). Da Worker1 von Thread die Methode start() erbt, kann auf Exemplaren von Worker1 diese Methode direkt aufgerufen werden. Die Klasse Worker2 hingegen *hat* bereits eine Superklasse, kann also nicht auch noch von der Klasse Thread abgeleitet werden. Die Alternative besteht darin, sie vom Interfacetyp Runnable abzuleiten. Ein Runnable hat selbst keine Methode start(), es benötigt also ein Thread-Exemplar,

damit seine run()-Methode ineinem neuen Betriebssystem-Thread gestartet werden kann. Normalerweise wird das Runnable dazu einem Thread-Exemplar in dessen Konstruktor mitgegeben.

Das vierte Beispiel (Test.java) demonstriert, warum es sinnvoll ist, dass ereignisverarbeitende Methoden eines Listeners schnell zurückkehren und wie man dies erreicht, indem man länger dauernde Aktivitäten in einen eigenen Thread auslagert.

In Zeile 14 wird aus einem Listener heraus eine langwierige Aktion gestartet: Während diese ausgeführt wird, ist der AWT-Event Dispatch Thread blockiert, d.h. Ereignisse die in dieser Zeit am GUI auftreten, werden erst ausgeführt, nachdem die langwierige Aktion beendet ist. Probieren Sie es aus, indem Sie das Fenster vergrößern oder verkleinern (es erfolgt keine Layout-Neuberechnung) oder zu schließen versuchen. Kommentieren Sie hingegen die Zeile 14 aus und entkommentieren Zeile 15, wird die langwierige Berechnung in einem neuen Thread gestartet und das GUI bleibt reaktiv. Vergleichen Sie bitte das Verhalten...

Die Zeilen 27-31 demonstrieren das Erzeugen eines neuen Threads, dem ein Runnable übergeben wird. Dieses Runnable ist hier ein Exemplar einer anonymen blocklokalen Klasse, die Runnable implementiert. Zu anonymen blocklokalen Klassen siehe auch den entsprechenden Abschnitt in bsp_11_awt.

Die Zeilen 33 und 34 demonstrieren, dass ein neuer Betriebssystem-Thread erst durch Aufruf von start() auf einem Thread-Exemplar gestartet wird: Wenn Sie Zeile 33 aus- und Zeile 34 entkommentieren, wird *kein* neuer Thread gestartet, d.h. die langwierige Aktion findet wieder im EDT statt.

Zeile 32 setzt die Priorität für das Thread-Exemplar auf den Standardwert. Dies ist sinnvoll, weil ein neu erzeugter Thread immer die Priorität des Threads erbt, aus dem er erzeugt wurde. Das wäre in diesem Fall der EDT, der eine relativ hohe Prioität hat, was für den neuen Thread i.A. nicht erwünscht ist.

Sehen Sie sich zum Thema "Threads" bitte auch den entsprechenden Abschnitt in der FAQ an, insbesondere die Erläuterungen zur Bedeutung von "synchronized".

Package bsp_15_ueberladung

Dieses Beispiel demonstriert mehrere im Zusammenhang mit der Auflösung von Überladung wichtige Sachverhalte.

- Aufruf A: Der Compiler kennt nur Deklarationstypen. Deshalb interessiert ihn bzgl. der Variablen f auch nicht, ob diese vielleicht zur Laufzeit eine Forelle referenzieren wird. Der Deklarationstyp von f ist Fisch. Und daher ist die Methode "2" für diesen Aufruf *nicht* passend. Passend zum Aufruf ist nur die Methode "1", ein Aufruf dieser Methode wird in den Bytecode geschrieben.
- Aufruf B: Der Compiler kennt nur Deklarationstypen. Die Variable s2 hat ebenso wie s1 den Deklarationstyp Super. In den Bytecode wird also wie bei Aufruf A ein Aufruf der Methode "1" geschrieben. Zur Laufzeit allerdings stellt die VM fest, dass s2 ein Exemplar der Klasse Sub referenziert, und schaut, ob in der Klasse Sub die Methode "1" überschrieben wurde. Dies ist der Fall: Die Methode "6" überschreibt die Methode, deren Aufruf im Bytecode steht, also wird Methode "6" ausgeführt.
- Aufruf C: Der Compiler kennt nur Deklarationstypen. Die Variable s1 hat den Deklarationstyp Super. Deswegen finden bei der Auflösung der Überladung auch nur Methoden Berücksichtigung, die für den Typ Super deklariert sind. Die Methode "5" wird also nicht berücksichtigt, und wenn sie von den Parametertypen noch so gut passen würde. Passend zum Aufruf ist nur die Methode "3", ein Aufruf dieser Methode wird in den Bytecode geschrieben.
- Aufruf D: Der Compiler kennt nur Deklarationstypen. Die Variable s2 hat ebenso wie s1 den Deklarationstyp Super. In den Bytecode wird also wie bei Aufruf C ein Aufruf der Methode "3" geschrieben. Zur Laufzeit allerdings stellt die VM fest, dass s2 ein Exemplar der Klasse Sub referenziert, und

schaut, ob in der Klasse Sub die Methode "3" überschrieben wurde. Dies ist *nicht* der Fall, also wird die Methode "3" ausgeführt. Die tatsächlichen Typen der übergebenen Parameter spielen bei Java zur Laufzeit *keine* Rolle (die Auflösung von Überladung erfolgt nur durch den Compiler und ist dann abgeschlossen), also kommt auch zur Laufzeit die Methode "5" nicht zum Zuge.

- Aufruf E: Der Compiler kennt nur Deklarationstypen. Die Variable s1 hat den Deklarationstyp Super. Deswegen finden bei der Auflösung der Überladung auch nur Methoden Berücksichtigung, die für den Typ Super deklariert sind. Die Deklarationtypen der Parameter von Aufruf E sind Huhn|Forelle. Dazu passen die Methoden "1" (Huhn SUB-> Tier, Forelle SUB-> Fisch) und "2" (Huhn SUB-> Vogel, Forelle SUB-> Forelle). Der Compiler muss also entscheiden, welche der passenden Methoden die speziellste ist. Das tut er, indem er aus der Liste der passenden Methoden alle streicht, zu denen es eine speziellere gibt. Spezieller als eine Methode a ist eine Methode b genau dann, wenn für alle deklarierten Typen der Parameter von b gilt, das ihr Deklarationstyp Subtyp des Deklarationstyps des entsprechenden a-Parameters ist. Die Frage, welche von zwei Methoden spezieller ist, lässt sich also unabhängig von einem konkreten Aufruf beantworten! In diesem Fall stellt der Compiler fest, dass die Methode "2" spezieller als die Methode "1" ist, denn es gilt: Vogel SUB-> Tier und Forelle SUB-> Fisch. Die Methode "1" wird also gestrichen, damit ist nur noch "2" übrig und damit die speziellste zum Aufruf passende Methode, also wird ein Aufruf von "2" in den Bytecode geschrieben.
- Aufruf F: Der Compiler kennt nur Deklarationstypen. Die Variable s1 hat den Deklarationstyp Super. Deswegen finden bei der Auflösung der Überladung auch nur Methoden Berücksichtigung, die für den Typ Super deklariert sind. Die Deklarationtypen der Parameter von Aufruf F sind Huhn|Huhn. Dazu passen die Methoden "3" (Huhn SUB-> Tier, Huhn SUB-> Huhn) und "4" (Huhn SUB-> Huhn, Huhn SUB-> Vogel). Der Compiler muss also entscheiden, welche der passenden Methoden die speziellste ist. Das tut er, indem er aus der Liste der passenden Methoden alle streicht, zu denen es eine speziellere gibt. Spezieller als eine Methode a ist eine Methode b genau dann, wenn für alle deklarierten Typen der Parameter von b gilt, das ihr Deklarationstyp Subtyp des Deklarationstyps des entsprechenden a-Parameters ist. Damit "3" spezieller wäre als "4", müsste also gelten: Tier SUB-> Huhn und Huhn SUB-> Vogel. Die erste Bedingung ist nicht erfüllt, also ist "3" *nicht* spezieller als "4". Damit "4" spezieller wäre als "3", müsste gelten: Huhn SUB-> Tier und Vogel SUB-> Huhn. Die zweite Bedingung ist nicht erfüllt, also ist "4" *nicht* spezieller als "3". Damit kann keine der beiden Methoden gestrichen werden, es bleiben zwei gleich spezielle passende Methoden, somit ist der Aufruf F für den Compiler nicht auflösbar. →Compilerfehler.

Es empfiehlt sich, bei der Auflösung von Überladung stur nach dem vorgegebenen Algorithmus vorzugehen, also zunächst alle zum Aufruf passenden Methoden auszuwählen, wenn es mehrere sind, wirklich für jede passende Methode a zu schauen, ob es zu dieser eine speziellere ebenfalls passende Methode b gibt und wenn ja, a zu streichen. Nur so ist man vor fehlerhaften Schnellsch(l)üssen sicher, wie z.B. dem, dass bzgl. des Aufrufs F die Methode "4" passender sei, weil deren Parametertypen Huhn|Vogel "näher" an Huhn|Huhn seien als die Parametertypen Tier|Huhn der Methode "3", oder dem, dass Methode "4" spezieller sei als Methode "3", weil Huhn|Vogel "weiter unten in der Typhierarchie liege" als Tier|Huhn. Das ist aber alles völlig uninteressant!

Package bsp_16_arraystorexception

Dieses Beispiel demonstriert Java-Code, der vom Compiler akzeptiert wird, aber zur Laufzeit eine ArrayStoreException auslöst. Ursache des Problems ist die Entscheidung Suns, aus einer bestehenden Subtypbeziehung zwischen Basistypen eine entsprechende Subtypbeziehung zwischen Arraytypen zu folgern, also `T[]` als Subtyp von `S[]` zu sehen, wenn `T` Subtyp von `S` ist.

Package bsp_17_wildcards

Dieses Beispiel zeigt, was die Deklaration einer Variablen mit einem Wildcard-Typ für die Menge der potentiell von dieser Variablen referenzierten Objekte aussagt, also, welche Eigenschaften der Compiler bzgl. dieser Objekte als sicher annehmen kann und welche nicht, welche Operationen er also demzufolge auf der Variablen zulässt:

Nur zur Sicherheit sei angemerkt, dass die Tatsache, dass ich den Variablen liste1, liste2 und liste3 den Wert null zuweise, absolut nichts mit den Fehlermeldungen des Compilers zu tun hat! Der Wert null (die leere Referenz) ist gültiger Wert *jeden* Typs und in den jeweils auf die Zuweisung folgenden Zeilen interessiert es den Compiler nicht im mindesten, was die Variable tatsächlich zur Laufzeit referenzieren wird, er kennt lediglich den Deklarationstyp und trifft seine Entscheidungen *ausschließlich* aufgrund dieses Typs.

Durch die Beschränkung der Wildcard bei liste2 und liste3 wird letztlich eingeschränkt, welche Typparameter jemand bei der Instantiierung einer konkreten Liste verwenden darf, die einer dieser Variablen zugewiesen werden kann. Und daraus resultiert für den Compiler ein Wissen über die Objekte, welche in einer solchen Liste vorhanden sein können.

Der Variablen liste1 können Listen mit beliebigem Parametertyp zugewiesen werden, d.h. dass der Compiler über diesen Typ nichts weiß. Es könnte sich also z.B. um eine ArrayList<Tier> handeln, aber auch um eine ArrayList<Object>. Also kann der Compiler uns nicht erlauben, über die Variable liste1 *irgendetwas* in die Liste zu füllen. Und bzgl. Eventuell aus der List zurückgegebener Objekte weiß er auch nicht mehr, als dass sie mit Sicherheit den Typ Object haben werden, weshalb er auch nur eine Zuweisung von aus der Liste geholten Objekten an eine Variable dieses Typs zulässt.

Der Variablen liste2 können Listen mit allen Parametertypen zugewiesen, die Subtyp von Tier sind (wie immer schließt das Tier selbst ein). Der Compiler kann uns nach wie vor nicht erlauben, *irgendetwas* in die Liste zu füllen, auch kein Tier, denn die tatsächlich von liste2 referenzierte Liste könnte auch z.B. eine ArrayList<Vogel> sein, und in die kann ich ja nicht jedes Tier packen. Über eventuell aus der Liste zurückgegebene Objekte weiß der Compiler allerdings, dass ihr Typ mit Sicherheit Subtyp von Tier sein wird, weswegen er eine Zuweisung von aus der Liste geholten Objekten an die Variablen zulässt die einen Supertyp von Tier als Deklarationstyp haben.

Der Variablen liste3 schließlich können Listen mit allen Parametertypen zugewiesen, die Supertyp von Vogel sind (wie immer schließt das Vogel selbst ein). Der Compiler kann uns nun erlauben, Objekte aller Subtypen von Vogel in die Liste zu füllen. Hingegen kann er über aus der Liste zurückgegebene Objekte mit Sicherheit nichts mehr sagen, außer, dass sie vom Typ Object sein werden: Im Extremfall könnte es sich tatsächlich z.B. um eine ArrayList<Object> handeln. Deswegen lässt er nur eine Zuweisung von aus der Liste geholten Objekten an Variablen des Deklarationstyps Object zu.

Package bsp_18_wildcards

Hier haben wir ein nettes Beispiel „aus der Praxis“, welches immer wieder gerne genommen wird, wenn man begründen möchte, wofür man eigentlich nach *unten* beschränkte Parametertypen benötigt. Es handelt sich um eine Liste, die eine Methode hat, welche alle Elemente einer übergebenen Liste als eigene Elemente übernimmt, und eine weitere Methode, welche alle eigenen Elemente in eine übergebene Liste füllt. Ich hoffe, es ist selbsterklärend, warum die Typen der Parameter der beiden Methoden genau so beschränkt sein müssen, wie sie es sind.

Package bsp_19_fragile_base_class

Objekte der Klasse IntegerBehaelter haben eine Methode add(), mit der man ein Integer-Exemplar in den Behälter legen kann. Außerdem gibt es eine Methode addAll(), der man eine Collection<Integer> übergeben kann und die *alle* Integers der Collection in den Behälter legt. Diese Methode addAll() ist so implementiert, dass sie über die übergebene Collection iteriert und für jedes Integer die Methode add() des Behälters verwendet, um es in den Behälter zu legen („erste Implementierung“).

Nun hat ein anderer Programmierer eine neue Klasse von IntgerBehaelter abgeleitet. Objekte dieser Klasse AddierenderIntegerBehaelter aktualisieren bei jedem Hineinlegen eines Integers ein Attribut, welches die Summe aller Integers im Behälter enthält. Dazu hat der Programmierer von AddierenderIntegerBehaelter die Methode add() aus IntegerBehaelter entsprechend überschrieben. Und da auch die (geerbte) Methode addAll() bei einem AddierenderIntegerBehaelter per dynamischer Methodenwahl intern die *neue* add()-Methode verwendet, wird auch beim Aufruf von addAll() auf einem Exemplar von AddierenderIntegerBehaelter die Summe korrekt aktualisiert.

Wenn nun aber der Programmierer von IntegerBehaelter irgendwann beschließt, statt in addAll() selbst über die Collection zu iterieren, dies lieber direkt die interne ArrayList, in der die Integer-Werte gelagert sind, erledigen zu lassen – diese verfügt über eine entsprechende Methode – dann führt diese scheinbar harmlose Änderung („zweite Implementierung“) in der Superklasse IntegerBehaelter dazu, dass die abgeleitete Klasse AddierenderIntegerBehaelter nicht mehr korrekt arbeitet, sondern eine falsche Summe liefert. Denn nun verwendet addAll() ja add() nicht mehr,

Package bsp_20_quadrat_rechteck_mit_offener_rekursion

In der Übersetzungseinheit Rechteck1.java wurde aus Gründen maximaler Code-Wiederverwendung Rechteck von Quadrat abgeleitet. Der Preis, den man dafür zahlt, ist hoch: Die abgeleitete Klasse Rechteck ist in Wirklichkeit überhaupt keine Spezialisierung von Quadrat, denn die Menge der Quadrate ist eine Teilmenge der Menge der Rechtecke und nicht umgekehrt!

In Rechteck2.java wurde der umgekehrte Weg beschritten: Quadrat wurde von Rechteck abgeleitet, was unter dem Gesichtspunkt von Generalisierung/Spezialisierug korrekt ist. Diese Variante hat aber den Nachteil, dass zum einen Quadrat jetzt ein im Grunde überflüssiges Attribut hat, und zum anderen ein Quadrat nur dadurch ein Quadrat ist, dass man implizit eine neue Klasseninvariante eingeführt hat, nämlich, dass laenge und breite immer denselben Wert haben müssen. Um deren Einhaltung unter allen denkbaren Umständen muss sich der Programmier ab jetzt kümmern. Das ist bei diesem einfachen Beispiel vielleicht noch vertretbar, bei einem komplexeren Szenario kann das aber durchaus problematisch werden.

Rechteck3.java schließlich zeigt eine elegante Lösung, welche mit offener Rekursion arbeitet. Sowohl Quadrate als auch nichtquadratische Rechtecke sind von einer gemeinsamen abstrakten Superklasse abgeleitet, in der die beiden Methoden flaeche() und umfang() bereits voll ausprogrammiert sind, aber intern zwei neue abstrakte Methoden laenge() und breite() verwenden, die korrekt zu implementieren, den Subklassen auferlegt wird. Diese tun dies auf unterschiedliche Weise. Diese Lösung vermeidet die Nachteile der beiden ersten Varianten. Und als zusätzliches „Bonbon“ hält sie sich auch noch an die Regel, dass nach Möglichkeit nur von den „Blättern“ des Klassenbaums Exemplare existieren sollten, während Superklassen möglichst abstrakt sein sollten.

Package bsp_21_listener

Am Beispiel eines Wecksdienstes wird die Funktionsweise von Listenern gezeigt.

Es geht darum, einer Ereignisquelle mitzuteilen, was sie beim Eintreten „ihres“ Ereignisses tun soll. Dies passiert, indem man bei der Ereignisquelle Interessenten anmeldet. Tritt dann später das Ereignis ein,

werden diese Interessenten per Aufruf einer ganz bestimmten Methode benachrichtigt („Callback“). Um sicherzustellen, dass diese Methode auch vorhanden ist, dürfen nur solche Objekte als Interessenten angemeldet werden, die von einem bestimmten Typ sind, nämlich eben von dem Typ, der die Existenz der Methode beschreibt.

Im Beispiel ist die Ereignisquelle ein Weckdienst. Dieser hat eine Methode, mit der man Weckdienst-Kunden beim Weckdienst anmelden kann. WeckdienstKunde ist ein Interface, welches genau eine Methode deklariert, nämlich die Methode wecke(). Ein WeckdienstKunde ist also jemand, dem man die Nachricht wecke() senden kann. Und genau das tut der Weckdienst im Beispiel, wenn es sechs Uhr ist. Dabei übergibt er als Parameter ein Objekt, das zusätzliche Informationen enthält., nämlich eine Referenz auf den Weckdienst selbst und einen String mit der Uhrzeit. Diese Informationen stehen dann in der Methode wecke() des WeckdienstKunden zur Verfügung.

In der main()-Methode der Klasse Test werden bei einem Weckdienst zwei Kunden angemeldet, nämlich einmal ein Exemplar einer konkreten Klasse HotelGast und zum anderen ein Exemplar einer anonymen lokalen Klasse. Beide Klassen implementieren natürlich das Interface WeckdienstKunde.

Eine Abbildung dieses Beispiels auf ein entsprechendes Konstrukt des AWT könnte so aussehen:

der Weckdienst (Ereignisquelle)	ein Button (Ereignisquelle)
die Methode anmelden() des Weckdienstes	die Methode addActionListener() des Buttons
das Interface WeckdienstKunde	das Interface ActionListener
die Methode wecke() in WeckdienstKunde	die Methode actionPerformed() in ActionListener
das WeckEreignis als Parameter von wecke()	das ActionEvent als Parameter von actionPerformed()

Package bsp_22_inner_outer

Ein Exemplar einer *nichtstatischen* inneren Klasse ist stets einem Exemplar der umgebenden Klasse zugeordnet. Auf dieses hat es eine implizite Referenz, über welche es auf die Methoden und Attribute des „umgebenden Exemplars“ zugreifen kann. Dies gilt auch für private Methoden und Attribute. Ein solcher Zugriff wird im Beispiel in der Methode m() des Inner gezeigt (Zugriff auf x).

Außerdem sehen Sie dort den Fall, dass ein Attribut y des Exemplars der nichtstatischen inneren Klasse ein gleichnamiges Attribut y „seines“ Exemplars der umgebenden Klasse verdeckt und wie Sie dieses trotzdem ansprechen können.

In der main()-Methode der Klasse Test sehen Sie die spezielle Syntax, mit der Sie einem neuen Inner „sein“ Outer mitteilen können. Der bei weitem üblichere Fall ist aber, dass Inner-Exemplare aus einem Outer-Exemplar heraus erzeugt werden, welches dann automatisch das Outer wird, dem diese Inner-Exemplare zugeordnet sind.

Ein Exemplar einer *statischen* inneren Klasse hingegen ist keinem Exemplar der umgebenden Klasse zugeordnet und hat daher natürlich auch keine implizite Referenz auf ein solches. → Compilerfehler