

## Musterlösung 4

### Aufgabe 1 (Vererbung)

*Welche der folgenden Behauptungen sind korrekt? Erklären Sie jeweils in einem Satz!*

1) RICHTIG: Eine Unterklasse ist eine Spezialisierung einer Oberklasse.

Beispiel: Oberklasse geometrische Form, Unterklasse Dreieck. Das Dreieck ist eine Spezialisierung der geometrischen Form. Nicht verwirren lassen: Obwohl die Unterklasse spezieller ist, hat sie mehr Methoden, Attribute.

2) KORREKT: Methoden und Attribute der Oberklasse stehen in der Unterklasse zur Verfügung, wenn sie nicht als `private` deklariert wurden.

Vererbung bewirkt genau diese Bereitstellung. Mit der Einschränkung, dass Attribute und Methoden, die als `private` deklariert wurden, nur in der Klasse, in der sie definiert wurden sichtbar sind. Dies bedeutet übrigens nicht, dass Fehler auftreten, wenn in geerbten Methoden auf diese Attribute zugegriffen wird.

3) FALSCH: Eine Klasse kann in Java von mehreren Oberklassen erben. Mehrfachvererbung wird in Java nicht unterstützt, da sie zusätzliche Komplexität in die Sprache hineinbringen würden. Siehe Vorlesung zum Thema Namenskonflikte.

4) FALSCH: Vererbung wird in Java durch das Schlüsselwort `inherit` durchgeführt.

In Java deklariert man eine Oberklasse durch Angabe hinter dem Schlüsselwort `extends`. `class Foo extends Bar { }`

5) RICHTIG: Eine Schnittstelle kann in Java von mehreren Oberschnittstellen erben.

Wieso dies erlaubt ist, wird noch genauer in der Vorlesung besprochen.

6) RICHTIG: Bei der Vererbung werden auch Schnittstellen mitvererbt (mit allen Rechten und Pflichten).

Getreu dem Grundsatz "Die Unterklasse kann mehr" werden Schnittstellen an Unterklassen vererbt. Diese implementieren damit implizit alle Schnittstellen der Oberklasse.

7) RICHTIG und FALSCH (== FALSCH): Konstruktoren und statische Methoden werden in Java vererbt.

Konstruktoren werden in Java nicht vererbt, statische Methoden und Attribute aber schon.

8) **KORREKT:** Ist die Klasse X Oberklasse von Y, so darf an jeder Stelle, an der ein Ausdruck vom Typ X erwartet wird, auch einer vom Typ Y stehen.

Dies ist einer der zentralen Punkte der Vererbung.

9) **RICHTIG:** Klasse A ist mit Klasse B verträglich, wenn A Unterklasse von B ist.

Wieder "A kann mehr als B", damit ist A mit B verträglich.

*Was ist der Unterschied zwischen Vererbung bei Klassen und der Implementierung von Schnittstellen durch Klassen?*

Um den Unterschied zu verstehen, müssen die Begriffe Implementierung und Schnittstelle klar sein. Mit Schnittstelle bezeichnen wir nur das äußere Erscheinungsbild(d.h. den Namen und Parametertypen, die sogenannte Signatur, also z.B. void sort(int[] data)), ohne dies aber wirklich mit ausführbarem Programmcode unterfüttert zu haben. Die Implementierung hingegen umfasst diesen ausführbaren Anteil an konkretem Code (in diesem Fall z.B. { Arrays.sort(data); }). Vererbung ist jetzt die Übertragung aller Eigenschaften einer Oberklasse auf die Unterklasse, d.h. sowohl Implementierung, als auch Schnittstelle. Durch die Ableitung entsteht also eine neue Klasse die alle Funktionalität von der Oberklasse übernimmt.

Vergleichen wir dies mit der Implementierung von Schnittstellen: Deklariert eine Klasse mittels "implements Schnittstelle", dass sie eine Schnittstelle implementiert wird, dann bekommt sie keinen konkreten Quellcode hierdurch übertragen und muss aber die eingegangene Verpflichtungen erfüllen, gewisse Methoden zu implementieren.

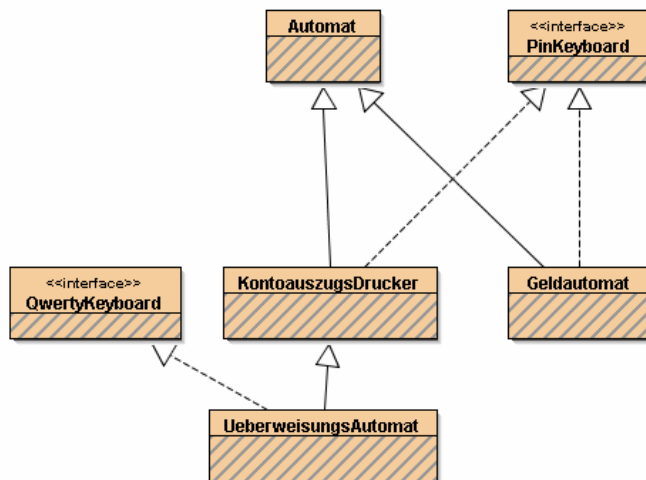
*Wieso sind beide Mechanismen getrennt wichtig?*

Schnittstellen dienen vornehmlich der Datenabstraktion, indem sie Implementierung und Schnittstelle trennen, während Vererbung der Wiederverwendung von Implementierungen dient. Insbesondere da Java keine Mehrfachvererbung besitzt, bekommt das Schnittstellenkonzept zusätzliche Bedeutung.

	Klasse selbst	Paket der Klasse	Unterklasse	anderswo
private	<b>J</b>	<b>N</b>	<b>N</b>	<b>N</b>
kein Modifizierer	<b>J</b>	<b>J</b>	<b>N</b>	<b>N</b>
protected	<b>J</b>	<b>J</b>	<b>J</b>	<b>N</b>
public	<b>J</b>	<b>J</b>	<b>J</b>	<b>J</b>

Wie man sehen kann, ist private eine wirklich sehr starke Einschränkung, die wahrscheinlich nur sehr selten wirklich gerechtfertigt ist (man verbietet ja damit die Wiederverwendung an allen Orten außer der Klasse selbst). Für saubere Datenabstraktion bieten sich hingegen eher an, Attribute oder Methoden ohne Modifizier stehen zu lassen oder mit protected zu versehen.

## Aufgabe 2 (Typverträglichkeit)



Finden Sie anschließend alle statischen Typfehler in folgender `main`-Methode.

```

41     a = u = k;
42     k = u;
43     g = u;
44     g = (GeldAutomat)u;
45     u = (UeberweisungsAutomat)g;
46     p = k; q = k;
47     q = (UeberweisungsAutomat)k;
48     p = q; q = p;
49     p = (PinKeyboard)q;
50     q = (QwertyKeyboard)p;
  
```

Zeile:	Fehler:
41	Kontoauszugsdrucker ist keine Unterklasse von Ueberweisungsautomat. Die Beziehung ist vielmehr umgekehrt, weshalb Zeile 42 korrekt ist.
43	Geldautomat und UeberweisungsAutomat haben zwar eine gemeinsame Oberklasse (Automat), aber stehen in keiner Unter-/Oberklassenbeziehung.
44,45	Deshalb ist auch trotz der Typumwandlung die Zeile 44 und 45 falsch. Es gibt keine Möglichkeit, dass diese Zuweisungen jemals gültig sind.
46	Typverträglichkeit bei Schnittstellen: Da die Klasse KontoauszugsDrucker nur die Schnittstelle PinKeyboard implementiert, ist die erste Zuweisung gültig, die zweite auf eine Variable vom Typ QwertyKeyboard jedoch nicht. Hierfür ist die Typumwandlung in Zeile 47 notwendig. Im Gegensatz zu Zeile 44 gibt es bei der Implementierung von Schnittstellen nicht die Möglichkeit, dass der Übersetzer erkennt, ob es sich um eine statisch ungültige Typumwandlung handelt. Wir wissen z.B. im konkreten Fall durch Zeile 42, dass in der Variablen k ein Exemplar der Klasse

	UeberweisungsAutomat gespeichert ist, welches die Schnittstelle QwertyKeyboard implementiert.
48, 49, 50	Ohne Typumwandlung (cast) sind die Zuweisungen in Zeile 48 natürlich nicht korrekt. Der Übersetzer wird anmerken, dass die Schnittstellen nicht miteinander in Beziehung stehen. Zulassen muss der Übersetzer die Typumwandlung in Zeile 49 und 50 aber dann doch, weil Klassen (wie z.B. der Übersetzungsautomat) ja durchaus beide Schnittstellen implementieren können.

### Aufgabe 3 (Polymorphie)

Vollziehen Sie nach, was in folgendem Java-Code passiert und geben Sie die Ausgabe des Programms ab. Sie sollten in der Lage sein zu erklären, was passiert.

```

7 class X {
8     int a = 4;
9     int get() { return a; }
10 }
11
12 class Y extends X {
13     static int a = 7;
14     int get() { return a; }
15     static void set(int x) { a = x; }
16     static void set(char c) { a = 2 * c; }
17 }
18
19 class Z extends Y {
20     static int b = 3;
21     int get() { return b + a; }
22     static int get(X x) { return x.a; }
23     static void set(int i) { a = 3 * i; }
24     static void set(X x, int i) { a = i; }
25     static void print(int args) { System.out.println(args); }
26     static void test() {
27         Z z = new Z();
28         print(z.a);
29         print(get(z));
30         print(((X) z).get());
31         z.set('c' - 'a' - 1);
32         print(get(z));
33         print(z.get());
34         Y y = z;
35         y.set(2);
36         print(z.get());
37         z.set(y, 0);
38         print(y.get());
39     }
40 }

```

Ausgabe:

Zeile	Ausgabe	Erklärung
28	7	Die Deklaration von a in Y verdeckt diejenige der Klasse X. Damit verweist z.a auf Z13.
29	4	Aufgerufen wird die Methode get der Klasse Z (Z22), das Attribut a wird statisch aufgelöst (es gibt keine

		Polymorphie bei Attributen) und verweist damit auf a auf Z8.
30	10	Der Methodenaufruf get() in Z30 wird trotz des statischen Typs des vorangehenden Ausdrucks (X) an die Exemplarmethode der Klasse Z verschickt (das ist ja genau Polymorphie). Aus a wird wieder ein Zugriff auf das statische a (Z13).
31		'c' - 'a' - 1 ist 1 (d.h. ein int) und wird der Methode set (Z23) übergeben (und nicht set (Z16)). Diese setzt dann, das statische a (Z13) auf den Wert 3.
32	4	Z31 hat das a (Z8), welches hier zurückgegeben wird nicht verändert.
33	6	Was sich aber ändert ist der Wert von z.get(), welcher auf a (Z13) zugreift. Dieser Aufruf ist übrigens in seiner Wirkung identisch mit Z30.
34		Gültige Zuweisung, da Z eine Unterklasse von Y ist.
35		Hier wird die Methode set (Z15) aufgerufen (Statische Methoden sind nicht polymorph) und der Wert von a (Z13) auf 2 gesetzt.
36	5	Damit wird jetzt durch z.get() 5 ausgegeben.
37		Es wird wieder nur auf a (Z13) zugegriffen und der Wert diesmal auf 0 gesetzt.
38	3	Die Ausgabe ändert sich damit auf 3.

Hier noch ein zusammenfassendes Beispiel:

```

class Foo {
    int instanceField = 1;
    static int staticField = 2;
    int instanceMethod() {
        return 3;
    }
    static int staticMethod() {
        return 4;
    }
}
class Bar extends Foo {
    int instanceField = 5;
    static int staticField = 6;
    int instanceMethod() {
        return 7;
    }
    static int staticMethod() {
        return 8;
    }
}

```

```

    }
}

```

Erstmal die offensichtlichen Fälle:

```

Foo f = new Foo();
f.instanceField == 1;
f.staticField == 2 && f.staticField == Foo.staticField;
f.instanceMethod() == 3;
f.staticMethod() == 4 && Foo.staticMethod() == f.staticMethod();

Bar b = new Bar();
b.instanceField == 5;
b.staticField == 6 && b.staticField == Bar.staticField;
b.instanceMethod() == 7;
b.staticMethod() == 8 && b.staticMethod() == Bar.staticMethod();

```

Der Zugriff auf Attribute und statische Methoden erfolgt immer aus dem statischen Kontext:

```

Bar b = new Bar();
((Foo) b).instanceField == 1;
((Foo) b).staticField == 2 && 2 == Foo.staticField;
((Foo) b).staticMethod() == 4;

```

Nur Exemplarmethodenaufrufe werden zur Laufzeit durch "Method-dispatch" an die richtige Stelle geleitet.

```

((Foo) b).instanceMethod() == 7 && 7 == b.instanceMethod()

```

Es ist also gar nicht mehr möglich von außen auf die Methode instanceMethod in Foo zuzugreifen. Nur innerhalb einer direkt erbenden Klasse (z.B. Bar und Bar2) wäre dies durch Verwendung von super.instanceMethod() erlaubt.

```

class Bar2 extends Foo {
    int instanceMethod() {
        return super.instanceMethod();
    }
}

Bar2 b = new Bar2();
b.instanceMethod() == 3

```

Mit diesem Wissen, kann man sich schon beinahe denken, wie man einen Konstruktor aus der Oberklasse aufruft (mittels super()).

```

class Foo {
    Foo(){ /* ... */ } // Konstruktor von Foo
}

class Bar extends Foo {
    Bar(){ super(); /*...*/ }
}

```

#### Aufgabe 4 (Ausnahmebehandlung)

- 1) FALSCH: Alle Ausnahmen, die eine Methode meldet (direkt oder indirekt), aber nicht behandelt, müssen im Methodenkopf deklariert werden.

Laufzeitausnahmen (d.h. alle Untertypen der Klasse RuntimeException) müssen nicht deklariert werden.

- 2) FALSCH: Fehler in Konstruktoren sollten nicht mit Hilfe von Ausnahmen behandelt werden.

Da Konstruktoren keinen frei verwendbaren Rückgabewert haben (sie geben ja das neue Exemplar zurück), gibt es gar keine andere Möglichkeit als Ausnahmen einzusetzen.

- 3) RICHTIG: Der `finally`-Block in einer Methode wird auch dann ausgeführt, wenn diese wegen einer nicht gefangenen Ausnahme abgebrochen wird.

Genau dies ist die Aufgabe des `finally`-Blocks.

- 4) RICHTIG: Eine in `main` aufgetretene, aber nicht behandelte Ausnahme führt zum Programmabsturz.

Die Java Virtual Machine behandelt die Ausnahme (durch Ausgabe des Aufrufkellers und der Textnachricht der Ausnahme) und beendet das Programm.

- 5) FALSCH: Durch Deklaration der Ausnahme aus 4) im Kopf von `main` lässt sich ein solcher Absturz verhindern.

Erstmal muss man natürlich die Ausnahme sowieso deklarieren, es sei denn es ist eine Laufzeitausnahme siehe 1). Aber auch dann ändert eine Deklaration nichts an der Tatsache, dass das Java Laufzeitsystem ja gar keine Ahnung davon hat, was es nun tun soll. Es bleibt nichts anderes übrig, als das Programm zu beenden.

- 6) FALSCH: Selbstdefinierte Ausnahmen werden bei der Übersetzung und zur Laufzeit anders behandelt als vordefinierte Ausnahmen aus `java.lang`. Nein. Selbstdefinierte Ausnahmen unterscheiden sich nur in der Tatsache, dass sie immer mittels `throw` ausgelöst werden müssen (eine `ClassCastException` wird z.B. im Kontrast hierzu durch eine ungültige Typumwandlung ausgelöst). Behandelt werden sie aber vollkommen identisch.

Was gibt folgendes Programm aus

```
A1: Done with 3
A1: Try failed: A1: NO FOO!
A1: Done with 4
A1: Done with 5
A1: Try failed: A1: NO BAR!
A1: Done with 6
A1: Done with 7
A1: Try failed: A1: NO FOO!
A1: Done with 8
Main: A2 threw exception: A1: 3 times
A1: Try failed: A1: NO FOO!
A1: Done with 0
A1: Done with 1
A3: A1: A1: Catch me
Main: An exception occurred:
```

```
java.lang.RuntimeException: A1: Catch me
    at U4.A4.ExceptionHandling.a1(ExceptionHandling.java:23)
    at U4.A4.ExceptionHandling.a3(ExceptionHandling.java:32)
    at U4.A4.ExceptionHandling.main(ExceptionHandling.java:46)
Main: I am going to rethrow it now!
Bye
Exception in thread "main" java.lang.RuntimeException: A1: Catch me
    at U4.A4.ExceptionHandling.a1(ExceptionHandling.java:23)
    at U4.A4.ExceptionHandling.a3(ExceptionHandling.java:32)
    at U4.A4.ExceptionHandling.main(ExceptionHandling.java:46)
```

*und wieso brauchen a1 und a2 keine Ausnahmedeklaration in der Signatur, a3 und main aber schon?*

Bei a1 und a2 werden Laufzeitausnahmen gemeldet, welche keiner Deklaration bedürfen.

*Beantworten Sie die Frage auf den Vorlesungsfolien ALP2-1.7 Seite 20: Könnte man anstelle eines Marken-Arguments auch ein Prozedur-Argument einsetzen? Begründen Sie kurz!*

Nein, eine Verwendung eines Prozedur-Aufrufs ist nicht möglich. Dies würde nämlich dazu führen, dass der Aufrufkeller bei jedem Fehler immer weiter wächst, da bei einem Prozeduraufruf die aufrufende Stelle immer gespeichert wird. Wird bei einem Fehler eine Prozedur aufgerufen, dann wird nach Beenden dieser wieder zurück an die fehlerhafte Stelle gesprungen. Damit ist nichts gewonnen.

### **Java Wiederholung - Folge 3**

Die Lösungen finden Sie unter <http://projects.mi.fu-berlin.de/w/bin/view/SS/JavaWiederholung>. Einsendungen bitte an oezbek@inf.