Polymorphismus und abstrakte Klassen

Zuweisungskompatibilität

```
Mitarbeiter Horst;
Horst = new Mitarbeiter();          // erlaubt

Manager Klaus_Baerbel;
Klaus_Barbel = new Manager();  // erlaubt
```

Natürlich sind die beiden obigen Anweisungen erlaubt. Erlaubt ist aber auch:

```
Mitarbeiter Otto;
Otto = new Manager();               // erlaubt
```

Otto ist vom Datentyp Mitarbeiter und mit einem Manager-Objekt verbunden. Dies ist deshalb erlaubt, weil die Klasse Manager mindestens die Methoden und Attribute der Klasse Mitarbeiter besitzt. Otto wäre daher auch als Mitarbeiter-Objekt voll funktionstüchtig. Allgemein gilt, dass ein Objekt einer abgeleiteten Klasse auch den Datentyp der Mutterklasse haben kann.

Anders sieht es im folgenden Beispiel aus:

```
Manager Susi;
Susi = new Mitarbeiter();           // nicht erlaubt
```

Hier besitzt das Objekt der Mutterklasse Mitarbeiter in der Regel weniger Methoden und/oder Attribute als die abgeleitete Klasse Manager. Susi wäre daher nicht auch als Manager-Objekt voll funktionstüchtig. Allgemein gilt, dass ein Objekt der Mutterklasse nicht vom Datentyp einer abgeleiteten Klasse sein darf.

Explizite Typkonvertierung

Erinnerung:
```
double value = 5.5;
int number = 0;
number = (int) value;
```

Ähnlich wie bei Variablen von einfachen Datentypen gibt es auch im Zusammenhang mit Objekten Typkonvertierungen. Zum Beispiel:

```
Mitarbeiter Klaus;
Klaus = new Manager();

Mitarbeiter Rolf;
Rolf = new Mitarbeiter();

Manager Klaus1 = (Manager) Klaus;          // Typkonvertierung ist ok

Manager Rolf1 = (Manager) Rolf;            // Typkonvertierung ist fehlerhaft
```

Im ersten Fall ist Klaus zwar vom Datentyp Mitarbeiter, ist aber mit einem Manager-Objekt verknüpft. Klaus1 ist eine Objektreferenz vom Datentyp Manager und darf daher natürlich mit dem Manager-Objekt verknüpft sein, mit dem auch Klaus verknüpft ist. Im zweiten Fall ist Rolf vom Datentyp Mitarbeiter und mit einem

Mitarbeiter-Objekt verknüpft. Rolf1 ist vom Datentyp Manager, kann sich aber nicht mit dem Rolf-Objekt verknüpfen, weil Rolf ein Mitarbeiter-Objekt ist und daher weniger Methoden und/oder Attribute hat als ein Manager-Objekt. Daher könnte Rolf1 nicht alles tun, was ein Manager tun kann, obwohl er Manager ist. Daher käme es zur Laufzeit in der Zeile

```
Manager Rolf1 = (Manager) Rolf;
```

zu folgender Fehlermeldung

```
Exception in thread "main" java.lang.ClassCastException:
Mitarbeiter at TypeCast.main(RuntimeError.java:8)
```

Mit den Zeilen

```
if (Rolf instanceof Manager) {
    Manager Rolf1 = (Manager) Rolf;
    }
```

hätten Sie vor dem Typecast *(Manager) Rolf* überprüfen können, ob Rolf ein Manager-Objekt ist oder nicht. Nur wenn Rolf ein Manager-Objekt wäre, wäre der Typecast zulässig.

Polymorphismus in der Vererbung

```
class Gehaltsauszahlung {

        public static void main(String[] args) {

                Mitarbeiter Otto = new Manager();

                Mitarbeiter Sabine = new Arbeiter();

                Mitarbeiter Karl = new Verkaeufer();

                . . .

                Otto.berechneGehalt();

                Sabine.berechneGehalt();

                Karl.berechneGehalt();

                . . .

                }

        }
```

Es wird immer die richtige Methode berechneGehalt() aufgerufen, auch wenn alle (Otto, Sabine, Karl) vom Datentyp Mitarbeiter sind. Das Laufzeitsystem von Java ermittelt die korrekte Methode über sogenannte vtables. Da die Funktionsweise von vtables etwas komplexer ist, verweise ich für besonders Interessierte auf die Literatur. Eine gute Beschreibung dieser vtables findet sich in Wikipedia.

Gibt es eine Methode in einer Klasse nicht, wird versuchet, die Methode der Mutterklasse (Basisklasse) aufzurufen.

Warum ist Polymorphismus wichtig?

Mittels Polymorphismus lassen sich Teile von Software automatisieren. Da alle Mitarbeiter denselben Datentyp haben, ließen sich alle Mitarbeiter eines Unternehmens in einem Array speichern. Zum Beispiel:

```
Mitarbeiter[] mitar;
mitar = Mitarbeiter[1500];
mitar[0] = Otto;
mitar[1] = Susi;
mitar[2] = Karl;
u.s.w.
```

Anschließend ließe sich z.B. in einer Schleife das Gehalt eines jeden Mitarbeiters berechnen:

```
for (int i=0; i<1500; i++) {
      mitar[i].berechneGehalt();
       }
```

Es gibt noch ein kleines Problem. Zur Laufzeit würde alles korrekt funktionieren und für jeden Mitarbeiter würde die richtige Methode berechneGehalt() aufgerufen werden. Der Compiler würde aber bei der Zeile

```
      mitar[i].berechneGehalt();
```

nach der Methode berechneGehalt() in der Klasse Mitarbeiter suchen, da mitar[i] vom Datentyp Mitarbeiter ist. Der Compiler weiß nicht, mit welchen Objekten (in unserem Fall vom Datentyp Verkaeufer, Manager oder Arbeiter) die jeweiligen Mitarbeitern zur Laufzeit verknüpft sind. Er kennt nur den Datentyp (Mitarbeiter) und gibt eine Fehlermeldung aus, weil er in der Klasse Mitarbeiter die Methode berechneGehalt() nicht findet. Es gibt zwei Lösungen:

1. berechneGehalt() in der Klasse Mitarbeiter leer implementieren

2. Mitarbeiter als abstrakte Klasse definieren

Abstrakte Klassen und abstrakte Methoden

```
abstract class Mitarbeiter {

        final double Basisgehalt = 1800.0;

        private int PersNr;

        private String Name;

        private int Plz;

        private String Ort;

        private String Strasse;
```

```
// Konstruktoren, Zugriffsmethoden

abstract double berechneGehalt();

}
```

Von abstrakten Klassen dürfen keine Objekte gebildet werden. Sie dient als Basisklasse für abgeleitete Klassen (in unserem Beispiel: Verkaeufer, Manager, Arbeiter) und kann abstrakte Methoden enthalten. berechneGehalt() ist im obigen Bespiel als abstrakte Methode definiert. Eine abstrakte Methode beginnt mit dem Schlüsselwort *abstract* und wird in der abstrakten Klasse nicht implementiert (auch nicht leer). Dem Compiler reicht das, weil er jetzt den Datentyp des Rückgabewertes, den Methodennamen und mögliche Parameter kennt.

Eine abstrakte Klasse kann von anderen abstrakten Klassen und von nicht-abstrakten Klassen geerbt werden. Werden sie von einer nicht-abstrakten Klasse geerbt, müssen alle abstrakten Methoden der abstrakten Klasse in der erbenden Klasse implementiert werden.