4.3 Vererbung

Ein wichtiges Prinzip bei der Softwareentwicklung ist die **Wiederverwendung** existierender Programmteile. Mit den Sprachmitteln, die wir bisher kennengelernt haben, gibt es dazu zwei Möglichkeiten:

- Prozeduren: Diese können benutzt werden, falls der Programmcode weitgehend gleich ist und sich nur durch einige Parameterwerte unterscheidet.
- „copy/paste/edit“: Kopiere den Programmteil und modifiziere diesen. Dies mag für die Bezahlung nach „lines of code“ gut sein, ist aber äußerst schlecht für die Lesbarkeit und Wartbarkeit von Programmen.

Objektorientierte Programmiersprachen bieten hierfür eine weitere strukturierte Lösung an: **Vererbung**. Hierunter versteht man die Übernahme von Merkmalen einer **Oberklasse** A in eine **Unterklasse** B. Zusätzlich kann man in B neue Merkmale hinzufügen oder Merkmale abändern (überschreiben). Wichtig ist aber, dass man nur diese Änderungen in B beschreiben muss, was zu einer übersichtlichen Programmstruktur führt.

Abbildung 4.1: Vererbungsstruktur

Beispiel 4.8 (Pixel)**.**
Pixel (Bildschirmpunkte) sind Punkte und haben eine Farbe. Mittels Vererbung können wir die oben definierte Klasse `Point` zur Vererbung in Java so nutzen:

```
class Pixel extends Point {  // Pixel erbt von Point
  Color color;

  public void clear () {
    super.clear();
    color = null;
  }
}
```

Anmerkungen:

- „`extends Point`“: Dieser Zusatz bedeutet, dass alle Merkmale von `Point` in die Klasse `Pixel` vererbt werden. Damit ist `Pixel` eine **Unterklasse** von `Point` und besitzt die Attribute `x`, `y` (von `Point`) und `color`.

- „`clear()`“: diese Methode existiert auch in Point und wird vererbt, aber in `Pixel` wird diese redefiniert. Ist also `p` ein `Pixel`-Objekt, dann bezeichnet „`p.clear()`“ die neu definierte Methode.
- „`super.clear():`“ Falls Methoden überschrieben werden, ist die vererbte Methode gleichen Namens nicht mehr zugreifbar. Aus diesem Grund gibt es das Schlüsselwort „`super`“, mit dem man auf Methoden der Oberklasse verweisen kann, d. h. `super.clear()` bezeichnet die `clear`-Methode der Oberklasse `Point`.

Zu beachten ist hier der Unterschied zwischen **Überschreiben** und **Überladen**:

Überschreiben: Dies bezeichnet die Redefinition geerbter Methoden (gleicher Name und gleiche Parametertypen). Damit ist die geerbte Methode nicht mehr zugreifbar (außer über das Schlüsselwort `super`).

Überladen: Dies bezeichnet die Definition einer Methode mit gleichem Namen, aber *unterschiedlichen* Parametertypen (auch innerhalb einer Klasse). Damit sind weiterhin beide Methoden (je nach Typ der aktuellen Parameter) zugreifbar.

Beispiel 4.9 (Überladung)**.**

```
class O {
  int    id(int    x) { return  x; }
  double id(double x) { return -x; }
}
```

In dieser Klasse ist der Name `id` überladen. Wenn wir also für ein `O`-Objekt die Methode `id` aufrufen, dann hängt es vom Typ des Parameters ab, welcher der beiden Methoden aufgerufen wird:

```
new O().id(1)    ⇝ 1
new O().id(1.0)  ⇝ -1.0
```

Das Konzept der Überladung (overloading) ist dabei unabhängig von OO-Programmiersprachen. Beispielsweise existiert dieses Konzept auch in Ada, funktionalen Sprachen (z.B. Typklassen in Haskell) etc. Ein klassischer Fall von Überladung ist die Operation „`+`“:

- `1 + 3` : Addition auf ganze Zahlen
- `1.0 + 3.0` : Addition auf Gleitkommazahlen
- `"af" + "fe"`: Stringkonkatenation

4.3.1 Sichtweisen der Vererbung

Bei der Vererbung sollte man zwei konzeptionelle Sichtweisen unterscheiden:

Modulsichtweise: Durch Vererbung wird ein Modul (Oberklasse) erweitert bzw. modifiziert, d. h. Vererbung entspricht einer Programmmodifikation, wobei nur die Modifikation aufgeschrieben wird.

Typsichtweise: durch Vererbung wird ein **Untertyp** B (ein speziellerer Typ) der Oberklasse A definiert, d. h. B-Objekte können überall eingesetzt werden, wo A-Objekte verlangt werden: „B ist ein A“ (B is-a A), aber mit speziellen Eigenschaften.

Viele OO-Programmiersprachen unterstützen beide Sichtweisen, was aber nicht unproblematisch ist:

- Die Typsichtweise ist problematisch beim Überschreiben von Methoden, denn dadurch kann die Unterklasse semantisch völlig verschieden von der Oberklassen sein. Beachte: Falls B ein Untertyp von A ist, sollte jedes Element aus B auch ein Element aus A sein.
- Die **semantische Konformität** (ist jedes B-Element auch ein A-Element?) ist im Allgemeinen unentscheidbar. Aus diesem Grund wird die Konformität abgeschwächt zu einer leichter überprüfbaren Konformität:

  > B heißt **konform** zu A, falls jedes B-Objekt überall anstelle von A-Objekten verwendbar ist, ohne dass es zu Typfehlern kommt.

Wie können sichere Typsysteme für OO-Sprachen aussehen, die die Konformität garantierten? Hierzu kann man sich z.B. mit folgender Approximation begnügen:

Definition 4.1 (Konformität)**.** *B* ist konform zu *A, falls für alle Merkmale m von A ein Merkmal m von B existiert mit:*

- *Falls m ein Attribut vom Typ τ in A ist, dann ist m ein Attribut vom Typ τ in B*
- *Falls m eine Methode in A mit Typ $\tau_1, \ldots, \tau_n \to \tau$, dann ist m eine Methode in B mit dem Typ $\tau'_1, \ldots, \tau'_n \to \tau'$ und*
  - *τ_i ist Untertyp von τ'_i (**Kontravarianz** der Argumente)*
  - *τ' ist Untertyp von τ (**Kovarianz** der Ergebnisse)*

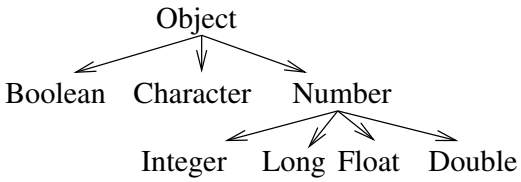
Pragmatische Lösungen in konkreten Programmiersprachen sind:

- Smalltalk: Hier erfolgt keine Konformitätsprüfung, da es keine statischen Parametertypen gibt. Somit führen Konformitätsfehler zu Laufzeitfehlern. Dieses Prinzip findet sich auch in anderen dynamisch getypten Programmiersprachen, wie z.B. Python, und wird auch als **duck typing** bezeichnet (“If it walks like a duck and it quacks like a duck, then it must be a duck.”). Dies bedeutet einfach, dass ein Objekt für ein anderes Objekt passend ist, wenn es die gleichen Merkmale bietet - was allerdings statisch nicht geprüft wird.
- Eiffel: Hier werden kovariante Argumenttypen verlangt, was manchmal natürlicher erscheint. Da dies allerdings zu Typfehlern führen kann, werden einige zusätzliche Prüfungen durch den Programmbinder vorgenommen.

- Java: Hier erfordert die Überschreibung von Methoden konforme (gleiche) Argumenttypen, sonst wird dies als Überladung interpretiert, wie wir oben schon am Beispiel einer Klasse gesehen haben. Dies ist jedoch eine vereinfachte Darstellung: in Wirklichkeit ist alles überladen und die Auswahl einer Methode erfolgt über einen „most specific match"-Algorithmus.

4.3.2 Polymorphie

Als Beispiel für eine Objekthierarchie betrachten wir die Klassen für primitive Datentypen in Java.



Die Klasse `Object` ist dabei die Urklasse aller Objekte, d.h. jede Klasse ohne `extends`-Klausel ist automatisch Unterklasse von `Object`. Dies ist ganz nützlich, um Polymorphie auszudrücken. Hierbei bedeutet **Polymorphie**, dass Objekte oder Methoden mehrere Typen haben können.

Beispiel 4.10 (Polymorphie in Java)**.** Wir betrachten ein Feld mit beliebigen Elementen.

```
Object[] a = new Object[100];
a[1] = new Point();
a[2] = new Integer(42);
a[3] = new Pixel();
```

Beachte: Alle Elemente in `a` haben den Typ `Object`. Dies bedeutet, dass z. B. `a[1].clear()` nicht zulässig ist. Falls dies gewünscht ist, müssen explizite Typkonversionen (**type cast**) eingefügt werden:

```
((Point) a[1]).clear()
```

Hierbei beinhaltet die Konversion `(Point)` eine Laufzeitprüfung, ob `a[1]` ein `Point`-Objekt ist. Somit führt z.B. der Aufruf

```
((Point) a[2]).clear()
```

zu einem Laufzeitfehler (`ClassCastException`).

4.3.3 Dynamische Bindung

Bei Vererbung ist zur Compilationszeit im Allgemeinen die Bindung eines Methodennamens zu einer Methode in einer Klasse nicht berechenbar. Dies ist eine Konsequenz der

Polymorphie. Die Zuordnung des Methodennamens zu einer Methode wird daher auch als **dynamische Bindung** von Methoden bezeichnet.

Beispiel 4.11 (Dynamische Bindung)**.**

Wir betrachten wir unsere Klassen `Point` und `Pixel`.

```
Point p;
⋮
p = new Pixel();    // zulässig, da Pixel auch ein Point ist
⋮
p.clear();          // Methode in Pixel, da p ein Pixel ist!
⋮
p.moveBy(1.0, 1.0); // Methode in Point, da Pixel diese ererbt
```

Operational bedeutet dies:

> Wenn eine Methode eines Objektes aufgerufen wird, dann suche in der Klassenhierarchie, beginnend bei der Klasse des aktuellen Objektes, nach der ersten Definition dieser Methode.

Hierbei gibt es allerdings eine Ausnahme: Ein **Konstruktor** ist eine Methode der Klasse, die **nicht** vererbt wird.

Betrachten wir hierzu das frühere Beispiel zur Definition von Konstruktoren:

```
class Point {
  ⋮
  Point (double x, y) { ... }
}
```

Wie kann man nun die Funktionalität des `Point`-Konstruktors in der Unterklasse nutzbar machen? Dies ist ebenfalls mittels `super` möglich, wobei `super` dann für den Konstruktor der Oberklasse steht (analog dazu steht `this` für den Konstruktor dieser Klasse):

```
class Pixel extends Point {
  ⋮
  Pixel (double x, double y, Color c) {
    super(x, y); // Aufruf des Point-Konstruktors mit x, y
    color = c;
  }
}
```

Dabei müssen `super(...)` bzw. `this(...)` die erste Anweisung im Rumpf sein. Falls diese fehlt, erfolgt ein impliziter Aufruf von `super()`, d. h. es wird der Konstruktor der

Oberklasse ohne Argumente aufgerufen. Somit ergibt sich die folgende Reihenfolge der Konstruktorabarbeitung/Objektinitialisierung:

1. Führe den Konstruktor der Oberklasse aus.
2. Initialisiere die Attribute entsprechend ihrer Initialwerte.
3. Führe den Rumpf des Konstruktors aus.

Diese Reihenfolge ist eventuell relevant bei Seiteneffekten in Methodenaufrufen von Konstruktoren.