4.2 Klassen und Objekte

Ein Modul in einer imperativen Sprache kann als ADT mit einem Zustand aufgefasst werden. Wenn so ein Modul importiert wird, wird nur genau eine Instanz (ein Modul existiert ja nur einmal) importiert. Dies hat den Nachteil, dass nicht mehrere Instanzen eines Moduls (z.B. mehrere Tabellen) gleichzeitig verwendet werde können.

Eine Lösung hierzu wurde schon früh in der Sprache Simula-67 vorgestellt: Fasse ein Modul als normalen Datentyp auf, von dem man mehrere Instanzen bilden kann, z. B. mehrere Variablendeklarationen vom Typ dieses Moduls. In diesem Fall spricht man von einer **Klasse**. Eine Klasse

- ist ein Schema für Module (in diesem Zusammenhang **Objekte** genannt),
- entspricht einem Datentyp, und
- unterstützt die Instanzenbildung.

Wir sprechen von einer **klassen- bzw. objektbasierten Sprache**, wenn die Programmiersprache die Definition und Instanzbildung von Klassen unterstützt. Eine **objektorientierte (OO-) Sprache** ist objektbasiert und unterstützt zusätzlich das Prinzip der Vererbung (vgl. Kap. 4.3).

Allgemein hat eine Klassenvereinbarung die Form

$$\texttt{class C \{}m_1\texttt{;} \ldots \texttt{;}m_k\texttt{;\}}$$

Anmerkungen:

- `C` ist der Name der Klasse, der wie ein Datentyp verwendbar ist, z. B. in einer Variablendeklaration:

  ```
  C x,y,z;
  ```

- $m_1, \ldots, m_k$ sind **Merkmale** der Klasse (in Java: **Mitglieder** bzw. **members**).
  Hierbei ist jedes m_i entweder
  - eine **Attributvereinbarung** (Java: **field**) der Form

    $$\tau\ x_1, \ldots, x_n;$$

  - oder eine **Methodenvereinbarung** der Form

    $$\tau\ \texttt{p}(\tau_1\, x_1, \ldots, \tau_n\, x_n)\ \{\ldots\}$$

    Dies entspricht im Wesentlichen einer Funktion oder Prozedur. Im Rumpf von `p` kann auf alle Merkmale der Klasse `C` zugegriffen werden.

- Instanzen der Klasse `C` können mittels

```
new C()
```

erzeugt werden. Hierdurch wird also eine neue Instanz der Klasse `C` erzeugt, d. h. ein neues Objekt, das alle Merkmale von `C` enthält.

- Der Zugriff auf Merkmale eines Objektes `x` geschieht mittels der **Punktnotation** `x.`m_i.
  - Falls m_i ein Attribut ist, dann bezeichnet dies den L- bzw. R-Wert des Attributs.
  - Falls m_i eine Methode ist, dann bezeichnet dies einen Prozeduraufruf mit `x` als zusätzlichem Parameter.

    Bei Abarbeitung des Rumpfes wird jedes Vorkommen eines Merkmals m_j im Rumpf der Methode durch `x.`m_j ersetzt.

    Der Aufruf einer Methode hat auch eine dynamische Sichtweise (z.B. wie in Smalltalk):

    „sende die Nachricht m_i an Objekt `x`“
- andere Auffassung: Objekt = Verbund + Operationen darauf + information hiding
- Zur Kontrolle des “information hiding” erlaubt Java die Spezifikation der Sichtbarkeit durch optionale Schlüsselwörter vor jedem Merkmal:
  - `public`: überall sichtbar
  - `private`: nur in der Klasse sichtbar
  - `protected`: nur in Unterklassen und Programmcode im gleichen Paket sichtbar („package“, vgl. Kapitel 4.6)
  - *ohne Angabe*: nur im Programmcode des gleichen Pakets sichtbar

Beispiel 4.3 (Klasse für Punkte im 2-dimensionalen Raum)**.**

```
class Point {
    public double x, y;

    public void clear() {
        x = 0;
        y = 0;
    }

    public double distance(Point that) { // distance to that Point
        double xdiff = x - that.x;  // (1)
        double ydiff = y - that.y;
        return Math.sqrt(xdiff * xdiff + ydiff * ydiff);
```

```
    }

    public void moveBy(double x, double y) {
        this.x += x;  // (2)
        this.y += y;
    }
}
```

Anmerkungen:

1. Im Rumpf von `distance` (1) bezeichnet `x` das Attribut des Objektes, für das die Methode `distance` aufgerufen wird. Dagegen bezeichnet `that.x` das entsprechende Attribut des Parameterobjektes.

2. Das Schlüsselwort `this` (2) bezeichnet immer das Objekt, für das der momentane Aufruf erfolgt, d.h. im Rumpf von `moveBy` bezeichnet `this.x` das Attribut dieses Objektes. Dagegen bezeichnet `x` den Parameter der Methode `moveBy`. In der Sprache Smalltalk wird das Schlüsselwort `self` statt `this` verwendet.

   In diesem Beispiel wäre die Verwendung von `this` vermeidbar durch Umbenennung der Methodenparameter. Im Allgemeinen ist `this` allerdings notwendig, um z.B. das eigene Objekt als Parameter an andere Methoden zu übergeben.

Anwendung der Klasse `Point`:

```
Point p;
p = new Point();
p.clear();
p.x = 80.0;
p.moveBy(1200.0, 1024.0);
// Nun ist p.x == 1280.0 und p.y == 1024.0
```

4.2.1 Operationale Semantik von Objekten

Wir können die operationale Bedeutung von Klassendeklarationen und Objektverwendungen relativ einfach durch Inferenzregeln definieren. Hierbei ignorieren wir aus Gründen der Übersichtlichkeit die Sichtbarkeitsregeln.

Klassendeklaration

$$\langle E \mid M \rangle \quad \texttt{class C } \{m_1; \ldots; m_k;\} \quad \langle E; C : class\{m_1, \ldots, m_k, E\} \mid M \rangle$$

Die Speicherung der momentanen Umgebung E ist notwendig, um bei späteren Methodenaufrufen die korrekte Deklarationsumgebung zur Verfügung zu haben.

Objektdeklaration

$$\langle E \mid M \rangle \quad \texttt{C x} \quad \langle E; x : (l, C) \mid M[l/\texttt{null}] \rangle \qquad \text{mit } l \in free(M)$$

Hier wird die Objektvariable `x` wie eine Referenz auf ein Objekt vom Typ `C` behandelt.

Objekterzeugung

$$\frac{E \vdash^{lookup} x : (l, C) \qquad E \vdash^{lookup} C : class\{\tau_1\, a_1, \ldots, \tau_n\, a_n, \langle Methoden \rangle, E'\}}{\langle E \mid M \rangle \quad \texttt{x = new C()} \quad \langle E \mid M[l/l'][l'/i_1] \ldots [l' + n - 1/i_n] \rangle}$$

wobei $l', \ldots, l' + n - 1 \in free(M)$ und i_j Initialwert vom Typ τ_j.

Somit wird Speicher für alle Attribute reserviert, aber nicht für die Methoden.

Zugriff auf ein Attribut

$$\frac{E \vdash^{lookup} x : (l, C) \qquad E \vdash^{lookup} C : class\{\tau_1\, a_1, \ldots, \tau_n\, a_n, \ldots\}}{\langle E \mid M \rangle \vdash^{L} x.a_i : M(l) + i - 1} \qquad \text{falls } M(l) \neq \texttt{null}$$

$$\frac{E \vdash^{lookup} x : (l, C) \qquad E \vdash^{lookup} C : class\{\tau_1\, a_1, \ldots, \tau_n\, a_n, \ldots\}}{\langle E \mid M \rangle \vdash^{R} x.a_i : M(M(l) + i - 1)} \qquad \text{falls } M(l) \neq \texttt{null}$$

Methodenaufruf $x.m(\ldots)$

Ein Methodenaufruf wird im Prinzip wie ein Prozeduraufruf behandelt, daher sparen wir uns eine detaillierte Definition. Es existieren aber die folgenden Unterschiede:

- Die Umgebung für die Methodenabarbeitung ist die Umgebung der Klassendeklaration, d. h. die Umgebung, die bei der Deklaration der Klasse gespeichert wurde (s.o.), einschließlich der Deklaration der Klasse selbst, damit auf alle Klassenmerkmale korrekt zugegriffen werden kann.
- Deklariere vor der Abarbeitung des Rumpfes in der lokalen Umgebung

  $$\texttt{this} : (l, C) \quad \text{falls } E \vdash^{lookup} x : (l, C) \text{ und } M(l) \neq \texttt{null}$$

- Ersetze im Rumpf alle sichtbaren Vorkommen von Merkmalen, d. h. alle Attribute und Methoden m_i dieser Klasse durch `this.`m_i.
- Die Parameterübergabe ist in Java immer der Wertaufruf, d. h. formale Parameter werden wie lokale Variablen behandelt.

4.2.2 Weitere Konzepte in Klassen und Objekten

Konstruktoren

Konstruktoren sind Methoden zur Initialisierung von Objekten, besitzen aber folgende Eigenschaften:

- Der Name eines Konstruktors ist der Name der Klasse, es existiert aber kein Ergebnistyp.
- Zusätzlich sind auch Parameter für Konstruktoren möglich und auch mehrere Konstruktoren sind für eine Klasse erlaubt, falls deren Parameter unterschiedlich sind.
- Die Konstruktoren werden nach der Objekterzeugung mittels `new` implizit aufgerufen, wobei die aktuellen Parameter die bei „`new`“ übergebenen Parameter sind.

Beispiel 4.4 (Konstruktoren)**.**

```
class Point {
  ...
  Point (double x, double y) {
    this.x = x;
    this.y = y;
  }
}

Point p;
p = new Point();          // nur Erzeugung
p = new Point(1.0, 2.0); // Erzeugung mit Konstruktoraufruf
```

Statische Attribute und Methoden

Falls das Schlüsselwort „`static`“ vor Attributen oder Methoden steht, dann gehört dieses Merkmal zur Klasse und nicht zu Objekten diese Klasse, d.h.

- es wird nur einmal repräsentiert (auch ohne Existenz von Objekten),
- der Zugriff erfolgt außerhalb der Klasse durch `<Klassenname>.m`.

Statische Merkmale werden auch **Klassenattribute und -methoden** genannt.

Beispiel 4.5 (Statische Merkmale)**.**

```
class Ident {
    public int nr;
```

```
    private static int nextid = 0;

    public static int currentnr () {
      return nextid;
    }

    Ident () { // Konstruktor
      nr = nextid;
      nextid++;
    }
}
```

Als Effekt erhält jedes `Ident`-Objekt bei seiner Erzeugung eine neue Nummer, die fortlaufend hochgezählt wird. Der Zugriff auf den aktuellen Zählerstand von außen erfolgt mittels `Ident.currentnr()`.

Konstanten und endgültige Methoden

Falls das Schlüsselwort `final` vor Merkmalen steht, dann ist dieses Merkmal nicht weiter veränderbar. Somit gilt:

- `final`-Attribute entsprechen Konstanten.

```
class Math {
  ⋮
  public static final double PI = 3.141592653589793;
  ⋮
}
```

- `final`-Methoden sind in Unterklassen nicht redefinierbar (Unterklassen werden später behandelt).
- `final class C {...}`: Alle Methoden sind implizit „`final`“ und es können keine Unterklassen gebildet werden.

`main`-Methode

Beim Start eines Java-Programms mittels

```
> java C <Parameter>
```

erfolgt ein Aufruf der Methode `main` aus der Klasse `C`. Zu diesem Zweck muss in der Klasse `C` die `main`-Methode wie folgt deklariert sein:

```
class C {
```

```
    ⋮
    public static void main(String[] args) {
      ...
    }
  }
```

Hierbei bedeuten:

`public`: Die Methode ist öffentlich bekannt.

`static`: `main` ist eine Klassenmethode und kann somit ohne Existenz eines Objektes aufgerufen werden.

`void`: `main` hat keinen Rückgabewert.

`args` ist die Liste der Aufrufparameter.

Reine OO-Sprachen

Konzeptuell ist es elegant, wenn alle Datentypen durch Klassen definiert sind („alles sind Objekte“). Sprachen mit dieser Sichtweise werden auch als *rein objektorientierte Sprachen* bezeichnet, wie dies z. B. in Smalltalk der Fall ist. Diese bieten die Vorteile

- einer einheitlichen Sichtweise (wichtig z. B. bei Generizität)
- und eines uniformen Zugriffs auf alle Objekte (alles sind Referenzen).

Allerdings existieren auch folgende Nachteile:

- Es existiert ein Overhead bei Grunddatentypen (`int`, `bool`, ...), da auch dies immer Referenzen sind.
- Die Sichtweise ist manchmal unnatürlich, z. B. bei der Arithmetik in Smalltalk: `3 + 4` ≈ sende an das Objekt „3“ die Nachricht „+“ mit dem Parameter „4“

Als Ausweg existieren *gemischte OO-Sprachen*, bei denen zwischen einfachen Grunddatentypen (`int`, `bool`, ...) und Klassen unterschieden wird (z. B. in C++ und Java). Als Nachteil ergibt sich jedoch, dass man Prozeduren, die Objekte (Referenzen) als Parameter verlangen, nicht mit einfachen Datentypen aufrufen kann. In Java ist dieses Problem gelöst, indem für jeden primitiven Typ eine entsprechende Klasse existiert, die Objekte dieses Typs repräsentiert.

Beispiel 4.6. Java-Klasse `Integer` für ganze Zahlen

```
Integer obj = new Integer(42); // erzeuge Objekt, das Zahl enthaelt
if (ob.intValue() == ...) ... // Wertzugriff
```

Java-Compiler erledigen die Konversion von primitiven Werten in Objekte und wieder zurück automatisch, sodass auch primitive Werte als Parameterobjekte notiert werden können (Auto(un)boxing).
Zusätzlich dienen diese Klassen auch als Sammlung wichtiger Prozeduren, die für die „tägliche Programmierung“ recht nützlich sind:

- `obj.toString()` $\rightsquigarrow$ Konvertierung zu Stringdarstellung, z.B. `"42"`
- `Integer.parseInt("42")` $\rightsquigarrow$ Konvertierung eines Strings in eine Zahl: `42`

Sichtbarkeit von Attributen

Bei unserer Definition der Klasse `Point` sind die Attribute `x` und `y` für alle sichtbar. Hierdurch ergibt sich das potenzielle Problem, dass jeder Benutzer diese ändern könnte. Als Lösung werden die Attribute als „`private`“ definiert, was aus softwaretechnischer Sicht in der Regel sinnvoll ist. Zum Beispiel sind in Smalltalk grundsätzlich alle Attribute immer privat (aber in Unterklassen sichtbar), sodass der Zugriff auf Attribute immer über Methoden erfolgt. Dies kann man natürlich auch in Java mit entsprechenden Sichtbarkeitsdeklarationen erreichen:

Beispiel 4.7 (Sichtbarkeitsdeklarationen in Java)**.**

```
class Point {
   private double x, y;

   public double getX() { return x; }

   public void setX(double newX) { x = newX; }
   ⋮
}
```

Als Effekt der Sichtbarkeitsdeklarationen ergibt sich:

- Der Zugriff auf ein Attribut geschieht über die entsprechende Methode („getter“-Methode).

  ```
  p.getX()
  ```

- Das direkte Setzen ist nicht mehr möglich.

  ```
  p.x = 80.0; ⇝ Compilerfehler
  ```

- Die Veränderung von Attributen wird durch Methoden gekapselt („setter“-Methoden).

```
        p.setX(80.0);
```