

2 Funktionen und Java

Nachdem im vorhergehenden Kapitel die grundsätzliche Vorgehensweise zur Erstellung von Java Programmen beschrieben wurde, wenden wir uns nun der Übertragung funktionaler Modelle in Java-Programme zu. Dabei beschränken wir uns (im Rahmen dieses Kapitels) ausschließlich auf die funktionalen Elemente. Außer Funktionskomposition, Verzweigung und Rekursion treten keine weiteren Elemente auf. (Pattern-matching läßt sich in Java nicht umsetzen.) Die Definition rekursiver Datentypen ist in Java sehr viel aufwendiger als in funktionalen Sprachen und muss deshalb auf einen späteren Abschnitt verlegt werden.

2.1 Grobstruktur von Java-Programmen

Bevor mit der Übertragung funktionaler Modelle begonnen werden kann, ist es erforderlich, wesentliche Bestandteile einer Java-Klasse etwas detaillierter zu diskutieren. Betrachten wir hierzu nochmals das Programm (1):

```
class HelloWorld {  
  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
  
}
```

Syntax einer Java-Klasse

Wie schon erwähnt wurde, setzen sich Java-Programme aus (im Allgemeinen mehreren) Klassen zusammen. In Backus Naur Form (BNF) läßt sich die Syntax einer Klasse im einfachsten Fall folgendermaßen angeben:

```
<Klasse> ::= 'class' <Klassenname> '{'  
    <Attributdefinitionen>  
    <Methodendefinitionen>  
    '}'
```

Eine Klasse wird somit stets durch das Schlüsselwort **class** eingeleitet, der Name der Klasse ist frei wählbar und in geschweiften Klammern steht der eigentliche Programmtext, der sich aus so genannten Attributen und Methoden zusammensetzt (dazu später mehr).

Lassen wir die Attributdefinitionen vorerst unberücksichtigt und betrachten nur die Methodendefinitionen: Das Konzept der Methode in objektorientierten Sprachen ist dem Funktionsbegriff der funktionalen Programmierung sehr ähnlich. Für das Verständnis der ersten Java-Programme dieser Handreichung genügt es, Methoden als Funktionen zu betrachten, die sehr eng an die Klasse, innerhalb derer sie definiert sind, gekoppelt sind.

Die `main`-Methode aus Programm (1), ist dabei ein erstes Beispiel einer Methode und gleichzeitig eine besondere Methode: Sie wird ausgeführt, wenn der Java-Interpreter `java` aufgerufen wird; sie stellt also gewissermaßen das Hauptprogramm eines Java-Programms dar (daher auch der Name). Der Bezeichner `main` ist dabei fest vorgegeben und ein sinnvolles Java-Programm enthält in der Regel eine Klasse, die eine `main`-Methode enthält.

Betrachten wir die Deklaration der Methode `main` genauer: Dem Funktionsnamen gehen die Worte **public** **static** **void** voraus, das Element `(String[] args)` folgt. Während Erläuterungen zu **public** und **static** in späteren Abschnitten gebracht werden, sollen das Schlüsselwort **void** und das Element `(String[] args)` bereits hier diskutiert werden: Bei funktionalen Programmen wird die Signatur einer Funktion, d.h. der Typ der Argumente

und der Typ des Rückgabewertes, aus dem Funktionsterm abgeleitet. In Java ist dies nicht möglich, die Signatur einer Methode muss angegeben werden! Dabei gilt: Der Rückgabotyp steht vor dem Funktionsnamen, die Argumente stehen in Klammern hinter dem Funktionsnamen, wobei jeweils vor den einzelnen Argumenten der Typ dieses Arguments steht; mit `String[]` wird somit der Typ des Argumentes `args` bezeichnet. Die Syntax einer Methodendefinition lässt sich somit (zunächst) folgendermaßen angeben:

```
<Methodendefinition> ::= 'public static' <Rückgabotyp>
                        <Methodenname> '(' <Argumentliste> ')' { '
                        <Methodenrumpf>
                        } '
<Argumentliste>      ::= <Typbezeichner> <Argumentbezeichner>
                        (',' <Typbezeichner> <Argumentbezeichner> )*
```

Die Definition der Syntaxvariable `<Methodenrumpf>` folgt später.

Der Datentyp `void`

Wie ist der Datentyp `void` aufgebaut? Man kann ihn vergleichen mit dem aus funktionalen Sprachen bekannten leeren Tupel, dessen Datentyp dort mit `unit` bezeichnet wird. Er besteht aus einem einzigen Element und wird in Java insbesondere dann verwendet, wenn eine Methode keinen „echten“ Rückgabewert besitzt, sondern nur den Zustand des Programms ändert oder ein Ergebnis am Bildschirm ausgibt, wie im Programm (1). Im Allgemeinen jedoch sind die Rückgabetypen von Methoden verschieden von `void`. Es sind diejenigen, die aus der funktionalen Programmierung geläufig sind: Ganze Zahlen (`int`), Wahrheitswerte (`boolean`), Zeichenketten (`String`), usw..

Methodenaufrufe

Die `main`-Methode des Programms (1) besteht aus einer einzigen Zeile

```
System.out.println("Hello World!");
```

Es handelt sich dabei um den Aufruf der Methode `println()` innerhalb der Klasse `System.out`. Das Argument dieser Methode ist eine Zeichenkette vom Datentyp `String` (`"Hello World!"`).

In Java gibt es zwei grundsätzlich unterschiedliche Arten von Methoden: Klassenmethoden und Objektmethoden. Vorerst arbeiten wir nur mit Klassenmethoden. Klassenmethoden (oder statische Methoden, daher das Schlüsselwort `static` in obigen Methodendeklarationen) sind nichts anderes als Funktionen, die einer bestimmten Klasse zugeordnet sind. Ihr Aufruf muss deshalb neben dem Methodenbezeichner auch den Namen der Klasse enthalten. Dieser steht vor dem Methodenbezeichner und ist von diesem durch einen Punkt getrennt. Es ergibt sich also folgende Regel für den Aufruf von Klassenmethoden:

```
<Methodenbezeichner> ::= <Klassenname> '.' <Methodenname>
<Methodenaufruf>      ::= <Methodenbezeichner> '(' <Argument> (',' <Argument>)* ')'
```

Die Java-Klassenbibliothek

Java stellt dem Benutzer eine enorme Zahl von Klassen mit entsprechenden Methoden zur Verfügung. Die Klasse `System.out` und die zugehörige Methode `println()` sind dafür Beispiele. Informationen über die vordefinierten Klassen und Methoden erhält man am einfachsten in der so genannten Java-API (Application Programming Interface) unter

<http://java.sun.com/j2se/1.5.0/docs/api/>

Dort sind zu jeder Klasse die entsprechenden Methoden und ihre Signaturen detailliert dokumentiert. Beispielsweise findet man zur Methode `println()`

```
public void println(String x)

    Print a String and then terminate the line. This method behaves as
    though it invokes print(String) and then println().

    Parameters:
        x - The String to be printed.
```

2.2 Klassenmethoden

Komposition statischer Methoden

Nachdem die elementarsten Bestandteile einer Java-Klasse bekannt sind, wenden wir uns nun der Erstellung von Java-Programmen durch Übersetzung bereits vorhandener funktionaler Modelle zu. Im Mittelpunkt steht zunächst die Komposition statischer Methoden.

Folgende Ocaml-Funktionen seien gegeben:

```
let quadrat x = x *. x;;

let quadrat_wurzel x = sqrt x;;

let diagonaleRechteck a b = quadrat_wurzel(quadrat a +. quadrat b);;

let diagonaleQuadrat a = diagonaleRechteck a a;;
```

Listing 2: Komposition von Funktionen in Ocaml

Es handelt sich also um Funktionsdefinitionen zur Berechnung der Diagonale im Rechteck. Die Übertragung in ein Java Programm ergibt:

```
class Figuren {

    public static double quadrat (double a) {
        return a*a;
    }

    public static double wurzel (double a) {
        return Math.sqrt(a);
    }

    public static double diagonaleRechteck (double hoehe, double laenge){
        return wurzel(quadrat(hoehe) + quadrat(laenge));
    }

    public static double diagonaleQuadrat (double seite) {
        return diagonaleRechteck (seite , seite);
    }

    public static void main (String[] args){
        System.out.println(diagonaleQuadrat(2.0));
    }
}
```

}

Listing 3: Komposition von Funktionen in Java

Bereits ein erster Vergleich der beiden Programme 2 und 3 zeigt, dass Java-Programme bei gleicher Funktionalität umfangreicher als funktionale Programme sind. Neben den grundsätzlichen Bemerkungen zum Erstellen von Funktionen aus Abschnitt (2.1) sind weitere Punkte erwähnenswert:

- Die Funktion `sqrt()` ist eine Klassenmethode der Java-Programmbibliothek `Math`. Wie in der Online-API von Java nachzulesen ist, enthält diese Klasse eine Vielzahl weiterer wichtiger mathematischer Funktionen, wie Funktionen für Zufallszahlen, trigonometrische Funktionen usw..
- Der Typ `double` entspricht dem Typ `float` in Ocaml und ist neben ganzen Zahlen (Typname: `int`), Wahrheitswerten (Typname: `boolean`) und einzelnen Zeichen (Typname: `char`) einer der so genannten Grundtypen von Java.
- Die Rückgabe von Funktionsresultaten erfolgt in Java über die Anweisung `return`.
- Die üblichen arithmetischen Operationen (+, −, *, /) können in Java sowohl auf Ausdrücke vom ganzzahligen Typ (`int`) als auch auf Fließkommazahlen `double` angewandt werden.
- Ein Java-Programm benötigt im Allgemeinen eine `main`-Methode um einen konkreten Aufruf durchzuführen. Der `main`-Methode entspricht in Ocaml die Befehlszeile innerhalb des Interpreters.

Die prinzipielle Syntax der Methodendefinition wurde bereits bei dem ersten Programm (1) angesprochen, wobei jedoch der Methodenrumpf ausgeklammert wurde. Aus dem „Hello World“-Programm, sowie den Funktionsdefinitionen (3) kann man folgern, dass ein Methodenrumpf aus Funktionsaufrufen und Rückgabeanweisungen bestehen kann. Dies führt zu folgender Regel (die natürlich später erweitert wird):

`<Methodenrumpf> ::= <Rückgabeanweisung> | <Methodenaufruf>`
`<Rückgabeanweisung> ::= 'return' <Ausdruck> ';' ;`

2.3 Verzweigungen

Im Gegensatz zum funktionalen Programmierparadigma lassen sich Verzweigungen in Java nur mit bedingten Anweisungen umsetzen; Pattern-matching gibt es in Java nicht!

Bedingte Anweisung

Rein strukturell unterscheidet sich die bedingte Anweisung in Java von der Fallunterscheidung in Ocaml nicht. Die bedingte Anweisung kann als dreistellige nichtstrikte Funktion angesehen werden, deren erstes Argument ein boolescher Ausdruck ist; das zweite und dritte Argument sind Ausdrücke identischen Typs. Die Auswertung der beiden letzten Argumente erfolgt nichtstrikt. (Hinweis: Eine Operation heißt **strikt**, wenn das Ergebnis der Operation undefiniert ist, falls eines der Argumente nicht definiert ist. Beispiel: In Ocaml erbt der Ausdruck `(True || 1=1/0)` den Wert `True`, obwohl der Ausdruck `1/0` nicht definiert ist. Die Operation `||` ist also in Ocaml **nichtstrikt**.)

Beispiel: Bei Paketen gibt es im Allgemeinen ein Preissystem, das den Preis in Abhängigkeit vom Gewicht des Paketes berechnet. Ein derartiges Preissystem könnte etwa folgendermaßen aussehen (Die angegebenen

Daten sind natürlich völlig frei erfunden; Übereinstimmungen mit real existierenden Preistabellen wären rein zufällig):

$$\text{Preis (Gewicht)} = \begin{cases} 7,00 \text{ Euro,} & 0 \text{ kg} < \text{Gewicht} \leq 5 \text{ kg} \\ 10,50 \text{ Euro,} & 5 \text{ kg} < \text{Gewicht} \leq 10 \text{ kg} \\ 14,00 \text{ Euro,} & 10 \text{ kg} < \text{Gewicht} \leq 20 \text{ kg} \\ \text{„Ablehnung wegen zu hohem Gewicht“} & \end{cases} \quad (1)$$

In einer funktionalen Sprache würde die Implementierung lauten:

```
let preis x = if (0.0 < x && x <= 5.0) then 7.0 else
               if (5.0 < x && x <= 10.0) then 10.5 else
               if (10.0 < x && x <= 20.0) then 14.0 else
               else failwith "Ablehnung";;
```

Die Implementierung in Java lautet:

```
class Paketpreise {

    public static double paketpreis(double gewicht){
        if (gewicht > 0 && gewicht <=5.0) return 7.0;
        else if (gewicht > 5.0 && gewicht <=10.0) return 10.5;
        else if (gewicht > 10.0 && gewicht <= 20.0) return 14.0;
        return -1;
    }

    public static void main(String[] args){
        System.out.println(paketpreis(7.0));
    }
}
```

Der Vergleich der beiden Programme zeigt, dass rein syntaktisch kein großer Unterschied besteht zwischen der bedingten Anweisung in Java und dem bedingten Ausdruck in Ocaml. Unterschiede ergeben sich lediglich bezüglich der Behandlung von Ausnahmen: In Ocaml lässt sich eine Ausnahme einfach durch einen failwith-Ausdruck erledigen. In Java gibt es natürlich ebenfalls die Möglichkeit solche Exceptions zu behandeln, jedoch ist ihre Anwendung etwas komplizierter und Exceptions werden zu einem späteren Zeitpunkt thematisiert. In obiger Java-Implementierung wurde einfach der Wert -1 zurückgegeben, um dem Benutzer das Auftreten einer Ausnahme zu signalisieren.

Die Syntax der bedingten Anweisung in Java lautet somit vorerst:

```
<Bedingte Anweisung> ::= 'if' <Bedingung> <Anweisung>
                        'else' <Anweisung>
<Anweisung>           ::= <Rückgabeeanweisung> | <Bedingte Anweisung>
<Rückgabeeanweisung> ::= 'return' <Ausdruck> ';' ;
```

Auch diese Regeln werden natürlich im Weiteren noch ausgebaut werden. Beachtenswert ist, dass die Regeln rekursiv formuliert sind: Die syntaktische Variable <Bedingte Anweisung> tritt auch in der <Anweisung>, die ihrerseits wiederum Element der <Bedingten Anweisung> ist, auf. Bedingte Anweisungen können also beliebig tief geschachtelt sein.

2.4 Rekursion

Mit den bisher zur vorgestellten Elementen von Java können bereits rekursive Methoden geschrieben werden. Dazu wählen wir wiederum rekursive Funktionen in Ocaml als Ausgangspunkt und übertragen diese in ein Java-Programm. Betrachten wir zunächst die (nichtlinear rekursive) Funktion zur Berechnung der Binomialkoeffizienten:

```

let rec binomial (n, k) = match (n, k) with
  |(0, k) -> 1
  |(n, k) when (k <= 0 || k >= n) -> 1
  |(n, k) -> binomial(n-1,k) + binomial(n-1, k-1);;

```

Die Übertragung in ein Java-Programm lautet dann:

```

class Binomialkoeffizienten {

    public static int binomialkoeff (int n, int k){
        if (n == 0 || k <=0 || k >= n) return 1;
        else return binomialkoeff (n-1, k) + binomialkoeff(n-1, k-1);
    }

    public static void main (String[] args) {
        System.out.println(binomialkoeff(10, 7));
    }
}

```

Prinzipiell besteht zwischen rekursiven Methoden in Java und rekursiven Funktionen in Ocaml somit nur ein syntaktischer Unterschied. Einige Kleinigkeiten sollten jedoch erwähnt werden:

- Der Vergleichsoperator „==“ in Java: Das einfache Gleichheitszeichen = wird in Java für die Zuweisung von Ausdrücken zu Variablen verwendet(siehe Abschnitt (3). Sollen dagegen die Werte zweier Ausdrücke auf Gleichheit überprüft werden, so ist das doppelte Gleichheitszeichen == zu verwenden. Direkt können in Java jedoch nur Ausdrücke, deren Werte Grundtypen (d.h. **boolean**, **int**, **double**, **char**) sind, verglichen werden (dazu später mehr).
- Um rekursive Methoden formulieren zu können ist die BNF-Regel für den Methodenrumpf in folgender Weise zu erweitern:

```

<Methodenrumpf>      ::= <Rückgabeeinweisung> | <Methodenaufruf>
                       | <Bedingte Anweisung>

```