

3.2 Datentypen und Methoden

Wie bei vielen höheren Programmiersprachen gibt es auch in Java *einfache* und *strukturierte Datentypen*. Die strukturierten Datentypen werden auch als Referenzdatentypen bezeichnet.

Einfache Datentypen:

boolean, *char* und die numerischen Datentypen: *byte*, *short*, *int*, *long*, *float*, *double*.

Referenz-Datentypen:

Alle *Array*-, *Class*- und *Interface*-Datentypen.

Einfache Datentypen werden so repräsentiert und abgespeichert wie im ersten Kapitel besprochen – *byte*, *short*, *int* und *long* als Zweierkomplementzahlen, *float* und *double* als Gleitkommazahlen, *boolean* durch ein Byte, *char* als ein Unicode-Zeichen.

Referenz-Datentypen werden als Referenz (Zeiger, Adresse) auf einen Speicherbereich, in dem die Komponenten abgelegt sind, repräsentiert.

3.2.1 Variablen

Variablen eines Datentyps sind Behälter für genau einen Wert eines einfachen Datentyps oder für genau eine Referenz auf ein Speicherobjekt.

```
int x = 42;
Object y = new Object();
```

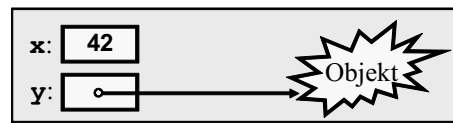


Abb. 3.3: Einfache und Referenz-Variable im Programm und im Speicher

Variablen müssen vor ihrer Benutzung *deklariert* worden sein. Dazu stellt man dem Namen einer oder mehrerer Variablen den Datentyp voran. Optional kann man eine Variable auch gleich mit einem Anfangswert initialisieren:

```
int x, y, z;
double r = 7.0 ;
boolean fertig = false ;
```

Variablen können gelesen und geschrieben werden. Ein Lesen der Variablen ist notwendig, wenn sie auf der rechten Seite einer Zuweisung auftaucht. Beispielsweise werden in

```
x = x+y;
```

die Variablen *x* und *y* gelesen, ihre Summe berechnet und als neuer Wert in die Variable *x* geschrieben. Den ersten schreibenden Zugriff auf eine Variable nennt man *Initialisierung*.

Nach ihrer Deklaration haben Variablen einen undefinierten Wert. Vor ihrer ersten Benutzung, d.h. bevor eine Variable zum erstenmal gelesen wird, muss sie *initialisiert* worden sein. Diese Vorgabe wird statisch, d.h. zur Übersetzungszeit, vom Compiler durch eine Datenflussanalyse überprüft. Dabei geht der Compiler auf „Nummer Sicher“. So würde direkt nach der obigen Deklaration die folgende Anweisung zu einer Fehlermeldung führen, obwohl das Ergebnis 0 weder von *x* noch von *y* abhängt:

```
if (x==x) return 0; else return y;
```

Der Compiler stellt nur fest, dass *x* für die Auswertung der Bedingung gelesen werden muss und dass im zweiten Ast der Anweisung auf *y* zugegriffen werden könnte.

Default-Werte

Für jeden Java-Typ existiert ein Standard-Wert, *auch default* genannt. Im Einzelnen sind dies:

- 0 für die numerischen Datentypen,
- **false** für *boolean*
- `\u0000` für *char* und
- **null** für alle Referenztypen.

Objekte oder Arrays werden durch expliziten Aufruf des **new**-Operators initialisiert. Dabei werden auch alle enthaltenen Komponenten initialisiert – wenn nicht anders festgelegt, mit dem Standard-Wert.

3.2.2 Referenz-Datentypen

Die strukturierten Datentypen werden in Java als *Referenz-Datentypen* bezeichnet, da man auf diese Daten nur indirekt über einen Zeiger zugreifen kann – eine *Referenz*. Eine solche Referenz kann entweder mit dem Default-Wert **null** initialisiert werden

```
int [] x = null;
```

oder es wird durch Aufruf des **new**-Operators ein entsprechendes Objekt geschaffen und ein Zeiger auf dieses neu angelegte Objekt zurückgegeben, wie z.B. in

```
Object p = new Object();  
int [] lottoZahlen = new int[6];
```

Im letzten Fall wird ein Array mit 6 Feldern angelegt, die alle mit 0 initialisiert sind.

Initialisierungen mit **null** sind gefährlich, weil formal zwar das Objekt, nicht aber seine Komponenten initialisiert werden. Sie hebeln daher die vorgenannte statische Überprüfung, ob eine Variable vor ihrer Benutzung initialisiert wurde, aus. So würde nach obiger „Initialisierung“ von *x* der folgende Code compilieren:

```
x[0]=x[0]+1;
```

Zur Laufzeit würde das Programm aber mit einer *NullPointerException* abbrechen, weil zwar *x* mit **null** initialisiert wurde, das array *x* und damit auch *x[0]* nicht existieren.

3.2.3 Arrays

Zu jedem beliebigen Datentyp T kann man einen zugehörigen *Array*-Datentyp $T[]$ definieren. Ein T -Array der Länge n ist immer eine von 0 bis $n-1$ indizierte Folge von Elementen aus T .

Array-Elemente:	17	-5	42	47	99	-33	42	19	-42	191
Indizes:	0	1	2	3	4	5	6	7	8	9

Abb. 3.4: Ein Array mit 10 Elementen vom Typ `int`

Es gibt zwei Möglichkeiten, Objekte eines Array-Datentyps zu erzeugen. Eine Möglichkeit besteht in der expliziten Aufzählung der Komponenten:

```
int[] int1Bsp = { 17, -5, 42, 47, 99, -33, 42, 19, -42, 191};
char[] char1Bsp = {'A', 'a', '%', '\\t', '\\\\', '\\', '\\u03a9'};
double[] double1Bsp = { 3.14, 1.42, 234.0, 1e-9d};
```

Die andere Methode ist, ein Array-Objekt mithilfe des `new`-Operators zu erzeugen. Dabei muss die Anzahl der Elemente, die das Array haben soll, angegeben werden. Der `new`-Operator reserviert Speicherplatz für ein neues Array-Objekt mit der gewünschten Zahl von Elementen und gibt eine Referenz auf dieses zurück.

Da die Speicherplatzreservierung, anders als z.B. in Pascal, erst zur Laufzeit des Programmes erfolgt, kann die Anzahl der Komponenten durch einen beliebigen arithmetischen Ausdruck bestimmt werden, dessen Wert erst zur Laufzeit ausgewertet wird. Man sagt, dass die Erzeugung von Arrays, allgemeiner von Objekten, *dynamisch* erfolgt. Bei dieser Gelegenheit erhalten die einzelnen Elemente des neuen Array-Objektes Standardwerte. Die Größe des Array-Objektes kann danach nicht mehr verändert werden.

```
char[] asciiTabelle = new char[256];
float[] tagesTemperatur = new float[365];
int orte = 100;
int[] distanzen = new int[orte*(orte-1)/2];
```

Wenn n die Anzahl der Komponenten eines Arrays ist, dann werden die einzelnen Elemente mit Indizes angesprochen, deren Wertebereich das Intervall 0 bis $n-1$ ist. Mit

```
tagesTemperatur[17] = tagesTemperatur[16]+1.5;
```

setzen wir die Temperatur des 18. Tages um 1.5 Grad höher als die des Vortages. (Da die Zählung mit 0 beginnt, ist `tagesTemperatur[17]` das 18. Arrayelement !)

Jedes Array-Objekt besitzt ein Feld `length`, das die Anzahl der Elemente des Arrays speichert. Daher kann man zum Durchlaufen eines Arrays Schleifen folgender Art benutzen:

```
for (int i=0; i < distanzen.length; i++){ distanzen[i]=0;}
```

Java kennt keine abkürzende Schreibweise für mehrdimensionale Arrays. Solche werden als Arrays aufgefasst, deren Komponenten selbst wieder einen Array-Datentyp haben. Der zum Datentyp `T[]` gehörende Array-Datentyp ist konsequenterweise: `T[][]`. Die Größe eines Arrays ist nicht Bestandteil des Typs. Daher sind auch nicht-rechteckige Arrays möglich:

```
int[] [] int4Bsp = new int[42][42];
int[] [] binomi = { { 1 }, { 1, 1 }, { 1, 2, 1 }, { 1, 3, 3, 1 } };
binomi[n][k] = binomi[n-1][k-1] + binomi[n-1][k];
```

3.2.4 Methoden

Methoden sind Algorithmen zur Manipulation von Daten und Objekten. Methoden umfassen und ersetzen die in anderen Programmiersprachen üblichen Begriffe wie *Unterprogramm*, *Prozedur* und *Funktion*. Methoden sind immer als Komponenten eines Objektes oder einer Klasse definiert. Eine Methodendeklaration hat die folgende Syntax:

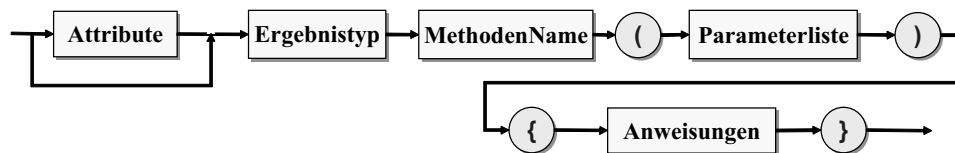


Abb. 3.5: Die Syntax von Methoden

Der Ergebnistyp kann ein beliebiger Java-Datentyp sein, dann handelt es sich um eine Funktion, die ein Ergebnis produzieren muss, oder es kann der *leere Datentyp* sein: `void`. Dann handelt es sich um eine Prozedur, die kein Ergebnis berechnet.

Jeder Parameter wird durch Angabe seines Datentyps und seines Namens definiert. Mehrere Parameterdefinitionen werden durch Kommata getrennt. Wenn die Parameterliste leer ist, muss man dennoch die öffnende und die schließende Klammer hinschreiben.

Auf die Parameterliste folgt ein *Block*, der aus einer in geschweifte Klammern „{“ und „}“ eingeschlossenen Folge von Java-Anweisungen besteht. Das Ergebnis einer Methode muss mit einer *return*-Anweisung zurückgegeben werden. Dies beendet die Methode.

In einer statischen Analyse überprüft der Compiler, dass garantiert jeder Zweig des Programms mit einer *return*-Anweisung beendet wird. Da `void`-Methoden keinen Wert zurückliefern, dürfen diese auf eine *return*-Anweisung verzichten.

Beispiel: Eine Funktion zur Berechnung der Fakultät (siehe S. 149). Diese Methode hat einen Parameter *n* vom Typ `int` und gibt ein Funktionsergebnis des gleichen Typs zurück.

```
int fak(int n) {
    if (n <= 0) return 1;
    else return n*fak(n-1);
}
```

Beispiel: Eine Prozedur zur Berechnung des größten gemeinsamen Teilers zweier Zahlen. Diese Methode hat zwei Parameter x und y vom Typ *int* und gibt kein Funktionsergebnis zurück, sondern schreibt ihr Ergebnis in das Standardausgabefenster.

```
void showGGT(int x, int y){
    System.out.print("ggT von "+ x + " und " + y + " ist: ");
    while (x != y)
        if ( x > y) x -= y;
        else y -= x;
    System.out.println(x);
}
```

Java-Anweisungen können auch Variablen deklarieren und ihnen einen Wert zuweisen. Diese Variablen sind in dem Block gültig, in dem sie definiert sind und in allen darin geschachtelten Blöcken. Es ist in Java nicht erlaubt in einem inneren Block eine Variable zu definieren, die den gleichen Namen trägt, wie eine Variable eines umgebenden Blockes.

Beispiel: Eine Prozedur zum Vertauschen von Elementen eines Array. Die Elemente an den Positionen i und k werden unter Verwendung der lokalen Variablen *temp* vertauscht. Es wird unterstellt, dass i und k gültige Indizes sind.

```
void swap(int[] a, int i, int k){
    int temp = a[i];
    a[i] = a[k];
    a[k] = temp;
}
```

Seit Version 1.5 des JDK können Methoden auch eine variable Anzahl von Argumenten haben. Der formale Parameter wird als Array aufgefasst und in der Deklaration durch „...“ gekennzeichnet. Eine Funktion, die beliebig viele Zahlen akzeptiert und deren Summe berechnet, können wir jetzt wie folgt programmieren:

```
int sum(int ... args){
    int sum=0;
    for(int i : args) sum+=i;
    return sum;
}
```

Ein Aufruf könnte beispielsweise in einer Ausgabeanweisung so erfolgen:

```
System.out.println(sum(12,42,-17,3,8,26));
```

Ausschließlich der letzte Parameter einer Methode darf eine variable Argumentanzahl haben, da sonst die aktuellen Parameter nicht eindeutig den formalen Parametern zugeordnet werden könnten.