Datum:

1 Grundlagen Programmierung

1.1 Datentypen

Bei der Programmierung in Java erinnern wir uns noch an diverse Datentypen. Also die Charakterisierung der möglichen Wertzuweisung einer Variable.

Typ	Beschreibung	Wertebereich / Beispiel
`boolean`	Boolescher Wert	`true`, `false`
`char`	einzelnes Zeichen	alle Unicode-Zeichen (Tastatur)
`byte`	eine ganze Zahl (max. 8 Bit)	$-2^7 \ldots 2^7 - 1$
`short`	eine ganze Zahl (max. 16 Bit)	$-2^{15} \ldots 2^{15} - 1$
`int`	eine ganze Zahl (max. 32 Bit)	$-2^{31} \ldots 2^{31} - 1$
`long`	eine ganze Zahl (max. 64 Bit)	$-2^{63} \ldots 2^{63} - 1$
`float`	Fließkommazahl (max. 32 Bit)	Beispiel: $3,14159f$
`double`	Fließkommazahl (max. 64 Bit)	Beispiel: $-1,79 * 10^{38}$
`String`	Zeichenkette (Wörter / Sätze etc.)	Beispiel: „Das ist ein String!“
`int[ ]`	ganzzahliges Feld (Array)	Beispiel: $3, 1, 4, 1, 5, 9$

Java unterscheidet zwischen **zwei Datentypen**. Zum einen gibt es die *primitiven Typen*. Dazu zählen z.B. `boolean`, `char`, ... , `double`. Zum anderen gibt es die *Referenztypen*. Als solche werden Typen bezeichnet, die entweder primitive Typen enthalten oder aus solchen zusammengesetzt werden. So zum Beispiel Objekte, Strings und Arrays.

1.2 Operatoren

In Java gibt es diverse Operatoren, die bei der Programmierung hilfreich sein können.

Java-Operator	Beschreibung	Anmerkung
+	Addition	
-	Subtraktion	
*	Multiplikation	
/	Division	Liefert den Quotienten von **x und y**. Sind beide Zahlen ganzzahlig, so auch der Quotient (z.B. 11/5 liefert 2).
%	Modulo	Divisionsrest (z.B 9%4 = 1)
++	Inkrement	`i++` entspricht dann `i+1`

Zusätzlich existieren noch weitere Operatoren die **Verknüpfung** zweier Variablen ermöglichen.

Java-Operator	Beschreibung	Anmerkung
=	Zuweisung	Der Variablen auf der linken Seite wird der auf der rechten Seite des =-Zeichen stehende Wert zugewiesen.
==	Vergleich	Ermöglicht den Vergleich von **primitiven Datentypen**. Liefert als Rückgabe `true` oder `false`.
<	Kleiner	Liefert `true` oder `false`.
<=	Kleiner gleich	
>	Größer	
>=	Größer gleich	
!=	Ungleich	Ermöglicht den Vergleich von **primitiven Datentypen**. Liefert als Rückgabe `true` oder `false`.
!	logisches NICHT	Kehrt die Wertzuweisung der nachfolgenden Variable für die nächste Operation um. Aus `true` wird `false`.
||	logisches ODER	Entweder die eine, die andere **oder** beide Bedingungen sind erfüllt.
&&	logisches UND	Es müssen beide Bedingungen erfüllt sein.

1.3 Klassendefinition

Wir erinnern und, dass es sich bei Java um eine sogenannte **objektorientierte Programmiersprache** handelt. Um ein Objekt überhaupt erzeugen zu können, benötigen wir einen *Bauplan*, der alle nötigen Informationen enthält. Diesen *Bauplan* bezeichnet man auch als **Klasse**.
Dabei gilt folgende Vorgabe:

```
(<Zugriffsart>) class <Bezeichner> (extends <Oberklasse>) {...}
```

Hierbei ist die Angabe der `<Zugriffsart>` notwendig, `extends <Oberklasse>` hingegen ist optional.

Beispiel:

```
public class Square{
  /**
  * Deklaration der Attribute
  * Ganzzahlige Attribute für Seitenlänge und Text-Variable für die Farbe werden
  * deklariert.
  **/
  private int length;
  private String color;
```

```
  /**
  * Methodendefinition
  * Konstruktor zur Erzeugung des Objekts hat den gleichen Namen wie die Klasse.
  **/
  Square(int side1){
    length = side1;
    color = "Red";
  }

  public double area(){
    return length*length;
  }
} //Ende der Klassendefintion
```

1.4 Methodendeklaration

Die eben angesprochene Klasse beinhaltet im Allgemeinen Methoden, also „Fähigkeiten“, die die erzeugten Objekte der Klasse besitzen.
Möchte man eine solche Methode deklarieren, so muss diese Deklaration die folgende Form haben:

```
(<Zugriffsart>) <Rückgabewert> <Bezeichner> (<Parameter>) {...}
```

Wie bei der Klasse ist die Definition der `<Zugriffsart>` verpflichtend. Die Angabe `<Parameter>` hat die Form `<Datentyp> <Bezeichner>`.
Die Definition des `<Rückgabewert>` bestimmt, welchen Datentyp die Methode bei Aufruf zurückliefert. Die möglichen Belegungen sind die unter 1.1 genannten, sowie weitere Datentypen. Die Angabe von `void` als Rückgabewert sagt aus, dass die Methode keine Rückgabe liefert.

Beispiel:

```
/**
* Öffentliche Methode hello gibt auf dem Bildschirm "Hallo XYZ" aus, wenn "XYZ" beim
* Aufruf übergeben wurde.
**/
public void hello(String name){
  System.out.println("Hallo " + name);
}

public double umfang(double radius){
  return 2*radius*3,14159;
}

/**
* Die Methode goToSleep hat keinen Rückgabewert und keine Parameter.
* Sie ruft nacheinander die Methoden undress, wash, brushTeeth und lieDown auf.
**/
public void goToSleep(){
  undress();
  wash();
```

```
    brushTeeth();
    lieDown();
  }
```

1.5 Variablendefinition

Innerhalb von Klassen, aber auch in Methoden benötigen wir Variablen, mit denen wir arbeiten können. Diese müssen zunächst deklariert werden. Auch hier gibt es eine Deklarationsvorschrift:

```
(<Zugriffsart>) <Typ> <Bezeichner> (= <Wert>)
```

Die direkte Wertzuweisung mittels = `<Wert>` kann, muss aber nicht, direkt bei der Variablendeklaration gemacht werden.
Bei dieser 'Wertzuweisung' ist wichtig zu beachten, dass **Referenztypen** im Allgemeinen mit dem `new`-Operator erzeugt werden müssen. Dies gilt nicht für den Referenztyp `String`.

Beispiel:

```
private int anzahl;
int tage = 15;
boolean healty;
\\
int[] counter = new int[Größe];
```

1.6 Zugriffsart

Bei der Definition bzw. Deklaration von Klassen, Methoden und Variablen wird immer nach der ominösen `<Zugriffsart>` verlangt. Diese gibt an, wer auf das Objekt und seine Methoden und Variablen zugreifen kann. Dabei gibt es die folgenden Unterscheidungen:

- **public**
  Innerhalb einer Klasse sind die Konstruktoren, Methoden und Variablen sichtbar. Sollen diese auch von Objekten außerhalb der Klasse verwendet werden, definiert man sie als `public`.
  Deklariert man eine Klasse als `public`, so können andere Klassen Instanzen dieser Klasse erzeugen.

- **private**
  Dem Gegenüber steht `private`. Diese Zugriffsart erlaubt den Zugriff nur innerhalb der Klasse selbst. Das bedeutet auch, dass z.B. Methoden oder Variablen, die als `private` deklariert wurden, für andere nicht sichtbar sind.

- **protected**
  Zusätzlich gibt es die Zugriffsart `protected`. Diese dritte Zugriffsart betrifft die Klasse, sowie alle derzeit existierenden und zukünftigen Subklassen.
  Auf als `protected` deklarierte Konstruktoren, Methoden und Instanzvariablen kann nur von Subklassen zugegriffen werden.

Befinden sich zwei Klassen im gleichen `package`, können diese jeweils auf die `protected` Bereiche der anderen zugreifen.

- **package (auch friendly oder default)**
  Der `default`-Modus tritt immer dann in Kraft,wenn keine ausdrückliche Zugriffsart angegeben wird.

`<Zugriffsart>`	**Beschreibung**
`public`	Der Zugriff ist immer möglich.
`private`	Der Zugriff ist nur innerhalb der Klasse möglich.
`protected`	Der Zugriff ist von Klassen innerhalb des gleichen Package möglich. Ebenso kann von Subklassen auf die `protected` Elemente zugegriffen werden.
`package`	Ein Zugriff ist innerhalb der Klasse und von anderen Klassen des gleichen Package möglich. Der Zugriff ist von einer Subklasse aus nicht möglich.

2 Grundlagen Struktogramme

Häufig ist es zu Beginn noch nicht klar, in welcher Sprache programmiert wird, so dass die ersten Überlegungen sehr universell sein müssen. Um das Ganze zu realisieren wurde eine Darstellungsform eingeführt, mit der die unterschiedlichen Anweisungen und Befehlsabfolgen allgemein verbildlicht werden können. Diese Darstellungsform wird **Struktogramm** (auch *Nassi-Schneidermann-Diagramm*) genannt.

2.1 Sequenz

Mehrere nacheinander ausgeführte Anweisungen, die in Java mit einem Semikolon (;) abgeschlossen werden, werden als **Sequenz** bezeichnet.

```
wash();
brushTeeth();
lieDown();
```

wash
brush teeth
go to bed

2.2 Fallunterscheidung

Unter einer Fallunterscheidung versteht man auch eine *bedingte Anweisung*. Diese gibt es zum einen **mit** und zum anderen **ohne** Alternative.

Mit Alternative:

```
if(<Bedinung>){
  <Anweisungen>
}
else{
  <Anweisungen>
}
```

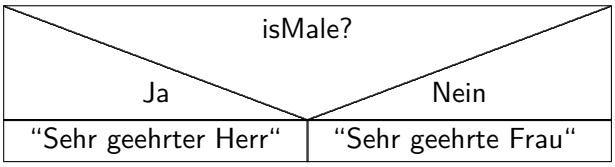
Ohne Alternative:

```
if(<Bedingung>) {
  <Anweisungen>
}
```

Beispiel mit Alternative:

```
if(gender == 'm') {
  System.out.println("Sehr geehrter
      Herr");
}
else {
  System.out.println("Sehr geehrte
      Frau");
}
```

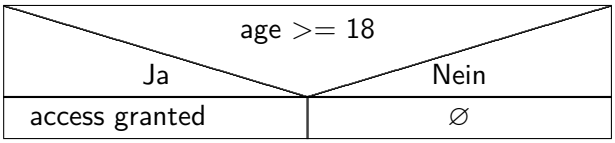
Mit Alternative



Beispiel ohne Alternative:

```
if(age >= 18) {
  access = true;
}
```

Ohne Alternative

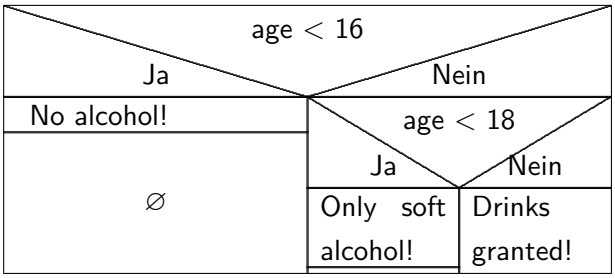


Bei der Fallunterscheidung kann es passieren, dass für verschiedene Bedingungen andere Anweisungen definiert werden. Dabei werden diese der Reihe nach überprüft, bis die erste erfüllt ist. Hierbei müssen aber die entsprechenden Bedingungen definiert werden.

Beispiel mehrere Bedingungen mit Alternative:

```
if(age < 16) {
  lightDrinks = false;
  hardDrinks = false;
}
else if(age < 18){
  lightDrinks = true;
  hardDrinks = false;
}
else{
  lightDrinks = true;
  hardDrinks = true;
}
```

Ohne Alternative



2.3 Mehrfachauswahl

Im Gegensatz zur Fallunterscheidung wird bei der der Mehrfachauswahl auf bestimmte Werte geprüft. So dass man hiermit beliebig viele Fälle ohne größeren Aufwand unterscheiden kann.
Die Java-Anweisung für die Mehrfachauswahl ist `switch`-Anweisung. Dabei muss aber beachtet werden, dass die zu überprüfende Variable vom Typ `byte`, `short`, `int` oder `char` sein muss.

Beispiel Mehrfachauswahl:

```
switch(<Variable>){
  case <Wert1>: <Anweisung 1>; break;
  case <Wert2>: <Anweisung 2>; break;
  ...
```

```
    default: <Anweisung n>; break;
  }
```

Die Anweisung `break` erzwingt das Verlassen der gesamten `switch`-Anweisung. Die nachfolgenden `case`-Bedingungen werden dadurch übersprungen.

Note = ?				
1	2	3	4	sonst
Mit Auszeichnung bestanden	gut bestanden	befriedigend bestanden	bestanden	nicht bestanden

2.4 Wiederholung

In Java hat man die Möglichkeit Anweisungen wiederholen zu lassen, ohne diese durch mehrfache Nennung im Programm-Code zu erzwingen. Dabei gibt es verschiedene Weisen, wie man dies tun kann.

2.4.1 Mit fester Anzahl

Möchte man beispielsweise etwas für eine bestimmte Anzahl wiederholen, so nutzt man die `for`-Anweisung.

```
for(<Init>; <Bedingung>; <Update>){
  <Anweisung>
}
```

Die geforderten Parameter haben die folgende Bedeutung:

- `<Init>` Deklaration einer ganzzahligen Zählvariable und der Zuweisung eines Anfangswerts (z.B. `int i = 0`)
- `<Bedingung>` Solange die Bedingung, abhängig von der Zählvariablen, erfüllt ist, werden die Anweisungen ausgeführt (z.B. `i < 5` - Die Sequenz wird 5 mal ausgeführt)
- `<Update>` Das Update der Zählvariablen erfolgt nach jedem Durchlauf entsprechend der angegebenen Zuweisung (z.B. `i++`)

Zu beachten ist, dass die `<Bedingung>` immer vor der Ausführung der Sequenz überprüft wird.

Beispiel Abbruchbedingung einbezogen:

```
/**
* Berechnet die Summe aller ganzen
    Zahlen von 0 bis 5.
*/
sum = 0;
for(i=0; i<=5; i++) {
  sum = sum + i;
}
```

Beispiel Abbruchbedingung ausgeschlossen:

```
/**
* Berechnet die Summe aller ganzen
    Zahlen von 0 bis 4.
*/
sum = 0;
for(i=0; i<5; i++) {
  sum = sum + i;
}
```

Die zugehörigen Struktogramme verdeutlichen den minimalen Unterschied in der `<Bedingung>`.

Von i:=0 bis 5 tue (wobei i jedes mal um1 erhöht wird)
Erhöhe den Wert der Summe um i

Von i:=0 bis 4 tue (wobei i jedes mal um1 erhöht wird)
Erhöhe den Wert der Summe um i

2.4.2 Mit Anfangsbedingung

Möchten wir erreichen, dass eine Anweisung oder eine Sequenz solange wiederholt, bis eine bestimmte Bedingung erfüllt wird, nutzt man die *Kopfgestützte* Wiederholung.

```
while(<Bedingung>){
  <Anweisungen>
}
```

Wir überprüfen also **bevor** die Sequenz ausgeführt wird, ob die Bedingung erfüllt ist. Dabei kann es also passieren, dass die Sequenz garnicht ausgeführt wird.

Beispiel kopfgesteuerte Wiederholung:

```
int pin = input();

while(pin != 0815){
  System.out.println("Fehlerhafte
      PIN!");
  pin = input();
}
```

Struktogramm

Eingabe einer ganzen Zahl
Solange eingegebene Zahl ungleich 0815
Bildschirmausgabe "Fehlerhafte PIN!"
Erneute Eingabe einer ganzen Zahl

2.4.3 Mit Endbedingung

Die in 2.4.2 erwähnte Wiederholungsanweisung prüft wie es der Name sagt bevor die Sequenz ausgeführt wird. Es kann nun aber natürlich auch mal passieren, dass eine solche Sequenz aber mindestens einmal ausgeführt werden soll, bevor die Abbruchbedingung überprüft wird. Um diese Anforderung zu erfüllen nutzt man die entsprechende *Fußgesteuerte* Wiederholung.

```
do{
  <Anweisungen>
} while(<Bedingung>)
```

Bei dieser wird die Sequenz einmal ausgeführt und die Bedingung wird **nach** jeder Wiederholung überprüft.

Beispiel fußgesteuerte Wiederholung:

```
do{
  System.out.println(number);
  number--;
} while(number > 0);
```

Struktogramm