Java – Datentypen und Kontrollstrukturen

Stephan Karrer

Autor: Stephan Karrer

Programmbeispiel

```
/* Einfache Ausgabe der Übergabeparameter */

public class Testübergabe {

  public static void main(String[] args) {
        if (args.length == 0)
              System.out.println("Es wurden keine Argumente angegeben");
        else for (int i=0; i<args.length; ++i) {
                    System.out.println("Argument"+(i+1)+": "
                                                    + args[i]);
        }
  }
}
```

- Für den Compiler muss der Name der Dateien mit dem Namen der public-Klasse übereinstimmen: **javac Testübergabe.java** ⇨ Testübergabe.class
- Für den Interpreter muss die Klassendatei eine entsprechende main-Methode haben: **java Testübergabe**
- Groß/Klein-Schreibung ist relevant ⇨ Plattform beachten

Grundlegendes zur Syntax

- Ein Java-Programm besteht stets aus einer oder mehreren Klassen (Daten und Funktionen *oder auch* Attribute und Methoden).
- Ausserhalb der Klassendefiniton können nur Kommentare und Anweisungen bzgl. der Namensräume (Paketstruktur), d.h. import- und package-Anweisungen, stehen.
- Innerhalb der Klassen werden für die Strukturierung wie üblich Blockstruktur und Funktionen genutzt.
- Innerhalb der Blöcke erfolgen (wie gewohnt) Variablendeklarationen und –zuweisungen, Berechnungen via Ausdrücke und Anweisungen.
- Java ist sehr variabel bzgl. der Schreibweisen
  - Reihenfolge der einzelnen Programmblöcke
  - Verwendung von Leerzeichen und Zeilenumbrüchen
  - Namensgebung

Basisdatentypen

`byte` (1 Byte)	`char` (2 Byte Unicode)
`short` (2 Byte)	`float` (4 Byte IEEE 754 Gleitkommazahl)
`int` (4 Byte)	`double` (8 Byte IEEE 754 Gleitkommazahl)
`long` (8 Byte)	`boolean`

- Einfach und vertraut (zumindest für C-Programmierer)
- Festlegungen maschinenunabhängig!
- implizite Typumwandlung (Casting) nur innerhalb der numerischen Typen, keine Typumwandlung zwischen boolean und den anderen Datentypen möglich
- ganzzahlige Datentypen sind alle mit Vorzeichen versehen (signed)
- Zeichen in Unicode-Darstellung ⇨ Internationalisierung!
- Für Variableninitialisierung existiert jeweils ein vordefinierter Wert

Wahrheitswerte

Bezeichner	boolean	
Literale	true, false	
Standardwert	false	
Operatoren	!	Negation
	&	Und mit vollständiger Auswertung
	&&	Und mit partieller Auswertung
	|	Oder mit vollständiger Auswertung
	||	Oder mit partieller Auswertung
	^	Exklusiv-Oder

Keine Konvertierung zu und von anderen Typen möglich !!

Zeichen

Bezeichner	char
Literale	'a', 'D', 'ö', '\u000D', '\u0022', ... '\b' Backspace (\u0008) '\t' Tabulator (\u0009) '\n' Neue Zeile (\u000D) '\f' Seitenumbruch (\u000C) '\r' Wagenrücklauf (\u000A) '\"' Doppeltes Hochkomma (\u0022) '\'' Einfaches Hochkomma (\u0027) '\\' Backslash (\u005C)
Standardwert	\u0000
Operatoren	

Beispiele	char c1 = 'A' ; char c2 = 'ä' ; // char c2 = '\u00E4'
	char c3 = '\n' ;

Ganzzahlen

Bezeichner	byte	short	int	long
Literale	1, -2, 8L, 123456789123455L 012, 0677, 0123456712123456L 0x12, 0x67FF 0b110011111, 0b1001		dezimal oktal hexadezimal binär	
Standardwert	0			
Operatoren	+, - +, -, *, /, % ++ --		Vorzeichen Addition, Subtraktion, Multiplikation, Division mit Rest, Modulo Post-und Preinkrement mit unterschiedlicher Semantik !! Post-und Predekrement mit unterschiedlicher Semantik !!	
Beispiele	int i = 1; int a = 0xFF; byte b = 12; long l = 7777L; Int k = 100_000 //Unterstrich wird durch Compiler ignoriert			

- Vorsicht bzgl. Unter/Überlauf
- Vorsicht bzgl. automatische Typumwandlung (Cast)

Bitweise Operatoren für Ganzzahltypen

Bit-Operatoren	~	Bitweise Komplement (Invertierung)
	|	Bitweise Oder
	&	Bitweise Und
	^	Bitweise Exklusiv-Oder
	>>	Rechtsschieben mit Vorzeichen
	>>>	Rechtsschieben ohne Vorzeichen
	<<	Linksschieben ohne Berücksichtigung des Vorzeichens

Gleitpunktzahlen

Bezeichner	float	double
Literale	3.14D, 3.14, 2f, .5F, 6. 1.2345E5	Standardnotation wissenschaftliche Notation zur Basis 10
Standardwert	0.0d bzw. 0.0f	
Operatoren	+, - +, -, *, /, % *(++* *--*	Vorzeichen Addition, Subtraktion, Multiplikation, Division, Modulo !! *Post-und Preinkrement mit unterschiedlicher Semantik !!* *Post-und Predekrement mit unterschiedlicher Semantik !!)*

- Für die Prüfung auf Über- bzw. Unterlauf stehen symbolische Konstanten aus den Hüllklassen Float bzw. Double zur Verfügung
- Spezielle mathematische Funktionen stehen über die Klassenbibliothek zur Verfügung

Vergleichsoperatoren für alle Datentypen

Operatoren	==	gleich
	!=	ungleich
	>	größer
	>=	größer oder gleich
	<	kleiner
	<=	kleiner oder gleich

Typanpassung (Cast)

- Bei den numerischen Datentypen existiert eine automatische Anpassung:

  byte ➔ *short* ➔ *int* ➔ *long* ➔ *float* ➔ *double* sowie

  char ➔ *int*

- Einschränkende Konvertierungen sind bei den numerischen Typen möglich, müssen aber explizit vorgenommen werden, z.B: *int n = (int) 3.14*

- Auch bei den automatischen Anpassungen ist Informationsverlust möglich (insbesondere long → float)

Variablen

```
public class Variablen {

    int i=0; final double KONSTANTE=1.2;

    public static void main(String[] args) {
        int i=2, k, l=1;
        System.out.println(i);  //liefert 2
        var zeichen='a';    //ab Java 10
        { double x=1.0, y;  y=x++;
          System.out.println(y);   //liefert 1.0
          //hier nicht möglich: int i;
        }   } }
```

- Variablendeklaration und Initialisierung kann auch in einem Schritt erfolgen
- Compiler stellt durch Datenflußanalyse sicher, dass jede Variable initialisiert ist
- Lokale Variablen müssen vom Programmierer vor dem ersten Zugriff initialisiert werden
- Instanzvariablen werden bei Bedarf automatisch initialisiert
- Nur einmalige Zuweisung soll möglich sein: ***final***
- Lokale Variablen können via ***var*** initialisiert werden, sofern der Compiler anhand der Zuweisung den Typ ableiten kann

Zuweisungsoperatoren für die numerischen Datentypen

Operator	Beispiel
=	int a, b ; a = b ;
+=	a += b ; // entspricht: a = a + b ;
-=	a -= b ; // entspricht: a = a - b ;
*=	a *= b ; // entspricht: a = a * b ;
/=	a /= b ; // entspricht: a = a / b ;
%=	a %= b ; // entspricht: a = a % b ;
&=	a &= b ; // entspricht: a = a & b ;
|=	a |= b ; // entspricht: a = a | b ;
^=	a ^= b ; // entspricht: a = a ^ b ;
<<=	a <<= b ; // entspricht: a = a << b ;
>>=	a >>= b ; // entspricht: a = a >> b ;
>>>=	a >>>= b ; // entspricht: a = a >>> b ;

Bedingte Anweisungen

if (*Bedingung*) { *Anweisungen* }

if (*Bedingung*) { *Anweisungen* } **else** { *Anweisungen* }

Erg **=** *Bedingung* **?** *Ausdruck1* **:** *Ausdruck2 ;*

```
if (x == 0)
  { y = 1;
    x = 1;
  }
```

Bei nur einer Anweisung im jeweiligen Zweig können die Block-Klammern entfallen

```
if (x == 0)
  { y = 1;
    x = 1;
  }
else
  { x = 2;
    y = 2;
  }
```

Bedingte Anweisungen: switch

```
switch (zeichen)
{   case '+': //add
      break;
    case '-': //sub
      break;
    case '/':
    case '*':
      System.out.println("cannot");
      break;
    default:
       System.out.println("illegal");
}
```

- Test ist bis Java 13 auf ***byte, short, int, char, String und Enums*** eingeschränkt
- Ist nicht abbrechend, deshalb optional ***break-***Anweisung
- Ab Java 14 bzw. Java 21 stehen erweiterte Varianten zur Verfügung

Switch: neues case-Label (Java 14)

```
String s = "zweig2";
switch (s) {
    case "zweig1", "zweig2"  -> System.out.println(1);
    //case "zweig3", "zweig2"  -> System.out.println(2); //not
allowed
    case "zweig3", "zweig4"  ->  { int i = 2;
                                   System.out.println(2); }
    default -> throw new IllegalArgumentException();
}
```

- Kein Fall-Through bei Verwendung des neuen Labels (case ... →)
- Zweige können kombiniert werden, wobei Konstanten-Duplikate nicht erlaubt sind
- Mehrere Anweisungen im Zweig müssen als Block geschrieben werden
  - Gültigkeit von Variablen im Block ist lokal für den Zweig !
- default-Zweig ist wie bisher optional

Switch-Expression (Java 14)

```
var i = switch (s) {     // i ist polymorph (Object,
Serializable, ...)
           case "zweig1", "zweig2"  -> "String1";        //String
           case "zweig3", "zweig4"  -> 2;             //int
           default -> throw new IllegalArgumentException();

     };   //Ausdruck abgeschlossen mit ;

System.out.println(switch (s) {
              case "zweig1", "zweig2"  -> "String1";
              case "zweig3", "zweig4"  -> 2;
              default -> throw new IllegalArgumentException();} );
```

- Der Zweig kann auch aus einem Ausdruck bestehen, dann muss aber auch jeder Zweig einen Wert (oder Exception) liefern
- Default-Zweig ist jetzt Pflicht !
- Auch hier kann der Compiler den Typ des Ergebnisses ableiten

Switch-Expression: yield

```
int n = switch (s) {
            case "zweig1", "zweig2"  -> { int k = "Hallo".length();
                                          yield k;
                                        }
            case "zweig3", "zweig4"  -> 2;
            default -> 1;
        };
```

- Ist ein Block für die Berechnung des Rückgabewerts notwendig, so muss der Wert per ***yield*** geliefert werden

Bedingte Schleifen

while (*Bedingung*) { *Anweisungen* } //abweisend

do { *Anweisungen* } **while** (*Bedingung*) //nicht abweisend

```
int sum (int a, int b){
    int s=0;
    while (a<=b) {
        s=s+a;
        a=a+1;
    }
    return s;
}
```

```
m1: while ( ... ){
  ...
  if (...) break;
  ...
  while (...){
    ...
    if (...) continue m1;
    ...
  }
}
```

Mit ***break***- und ***continue***-Anweisungen ist ein bedingtes Verlassen der Schleifen möglich

Iterierte Schleife

for (*Initialisierung* **;** *Bedingung* **;** *Iteration* **)** { *Anweisungen* }

```
/* Einfache Ausgabe der Übergabeparameter */

public class Testübergabe {

  public static void main(String[] args) {
        if (args.length == 0)
              System.out.println("Es wurden keine Argumente angegeben");
        else for (int i=0; i<args.length; ++i) {
                    System.out.println("Argument"+(i+1)+": "
                                                + args[i]);
        }
  }
}
```

- Selbstverständlich kann man die Schleifen schachteln
- Es muss nicht im klassischen Sinne gezählt werden, d.h. verwendete Datentypen und Iterationsschritt sind frei wählbar

For Each-Schleife

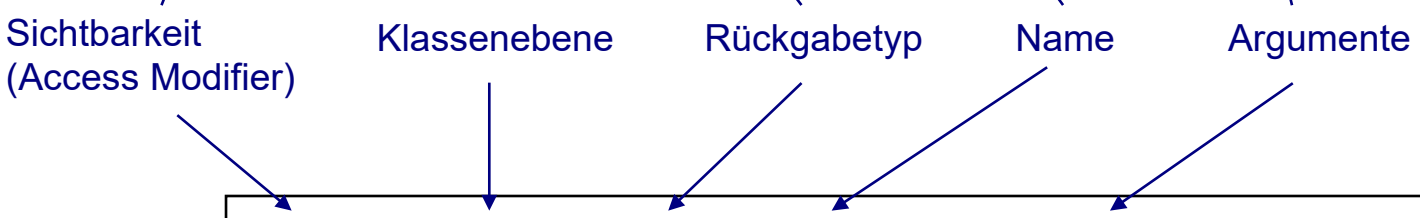
for (Laufvariable *: Wertemenge* **)** { *Anweisungen* }

```
public class Testübergabe {

   public static void main(String[] args) {

      for (String s : args) {
         System.out.println("Argument: "+ s);
      }
   }
}
```

- Schleife basiert intern darauf, dass die Wertemenge einen Iterator liefert
- Es ist in dieser Form nur Lesen ohne Schrittkontrolle möglich

Methoden

```
public double computePayment(double loanAmt,
                             double rate,
                             double futureValue,
                             int numPeriods) {
        // Berechnung
        return answer; }
```




```
public static void main(String[] args){
    // Berechnung
    return;  //optional
    // static void exit(int status)ist optional
}
```

Methodenaufruf mit variabler Parameterliste

```
public class Testübergabe {

   public static void main(String... args) {

      for (int i=0; i<args.length; ++i) {
          System.out.println("Argument"+(i+1)+": " + args[i]);
      }
   }
}
```

- Das letzte Eingabeargument einer Methode kann auch in Form einer variablen Liste eines Datentyps angegeben werden.
- Technisch gesehen ersetzt der Compiler dies gegen ein entsprechendes Array