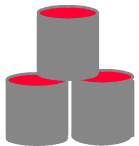
Grundlagen der Informatik
– Kern imperativer Sprachen –

Prof. Dr. Bernhard Schiefer

(basierend auf Unterlagen von Prof. Dr. Duque-Antón)

bernhard.schiefer@fh-kl.de
http://www.fh-kl.de/~schiefer



Inhalt

- Einführung
- Variablen und Zuweisungen
- Datenstrukturen
- Ausdrücke und Operatoren
- Kontrollstrukturen

Programmiersprachen

- Eine imperative (auch befehlsorientierte genannte) Programmiersprache besteht i.d.R. aus einer Folge von Zuweisungen (Anweisungen oder Befehlen).
- Die elementare Aktion „Zuweisung“ besteht aus
  - ⇨ einer Phase, in der ein Ausdruck ausgewertet wird, und
  - ⇨ einer Phase, in der das Ergebnis gespeichert wird.
- Auf diese Weise lassen sich komplexe Berechnungen als Folge dieser elementaren Aktionen beschreiben.
- Zur Organisation des Ablaufs der Berechnungen werden Kontrollstrukturen benötigt. Man unterscheidet:
  - ⇨ Blöcke
  - ⇨ Bedingungen
  - ⇨ Schleifen

Programmiersprache Java

- Java wurde um 1994 bei der Firma Sun (Gruppe um James Gosling) entwickelt. Ziele der Neuentwicklung waren insbesondere:
  - ⇨ Plattformunabhängige Sprache.
  - ⇨ Öffnung der Sprache für Netzwerkanwendungen und andere Systemfunktionen.
  - ⇨ Einfache Anwendung der Sprache im Gegensatz zu C bzw. C++
- Java ist frei verfügbar. Die für die Programmentwicklung benötigte Software und Dokumentation kann aus dem Internet bezogen werden
  - ⇨ Download Quelle: http://www.oracle.com/technetwork/java
  - ⇨ Um Java-Programme ausführen zu können, genügt das JRE (Java Runtime Environment).
- Um Java-Programme komfortabel entwickeln zu können, ist eine IDE (Integrated Development Environment) empfehlenswert (Beispiel: Eclipse)
  - ⇨ Ebenfalls empfehlenswert: Die Java-Online Dokumentation im HTML-Format http://docs.oracle.com/javase

Erste Schritte

- Installieren Sie bitte auf Ihrem Laptop oder Heimrechner das Java-JDK von Oracle, welches von der Homepage von Sun geladen werden kann.
  - ⇨ Sie finden es unter http:/java.sun.com
  - ⇨ Installieren Sie bitte auch die entsprechende HTML-Dokumentation.
- Das „Hello World“-Programm

```
// Programm HelloWorld
// Autor: Harry Hurtig

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

Kompilieren und Ausführen

- Compilierung aus der Kommandozeile:
  - ⇨ **javac** HelloWorld.java (im aktuellen Verzeichnis)
  - ⇨ Als Ergebnis wird (im aktuellen Verzeichnis) der Java-Byte-Code HelloWorld.class erzeugt.
- Die Ausführung des Programms erfolgt mit Hilfe des Kommandos
  - ⇨ **java** HelloWorld (ebenfalls im aktuellen Verzeichnis).
  - ⇨ Der zuvor produzierte Java-Byte-Code wird vom Interpreter ausgeführt.
  - ⇨ Dazu muss der Java-Interpreter **java** installiert und zugreifbar sein.
- In Java werden mehrere Einheiten dynamisch gebunden.
  - ⇨ Bei Bedarf zur Laufzeit holt sich der Java-Interpreter die benötigten (in Java-Byte-Code bereits compilierten) Dateien.
- Im Gegensatz dazu arbeitet die Programmiersprache C mit einem Compiler und erlaubt auch statisches Binden.

Bezeichner

- ... sind Namen für Variablen, Klassen (Objekte), Interfaces, Methoden (Funktionen) und Pakete. Sie können aus beliebig vielen Unicode-Buchstaben und Ziffern bestehen.
- Zusätzlich sind der Unterstrich _ und das Dollarzeichen $ erlaubt.
  Das erste Zeichen eines Namens darf keine Ziffer sein.
- Es wird zwischen Groß- und Kleinschreibung der Namen unterschieden.
  - ⇨ Java ist „case sensitive“
- Die Bezeichner dürfen nicht mit den Schlüsselwörtern der Sprache und den Literalen **true** / **false** und **null** übereinstimmen.
- Mit Hilfe der Java-Methode **Character.isJavaIdentifierStart (char)** kann getestet werden, ob der verwendete Buchstabe als erstes Zeichen erlaubt ist. In diesem Fall liefert die Methode den Wert true.
- Mit Hilfe der Java-Methode **Character.isJavaIdentifierPart (char)** kann getestet werden, ob der verwendete Buchstabe als Zeichen erlaubt ist.

Schlüsselwörter

- Die folgende Liste enthält die für Java reservierten Schlüsselwörter, die daher nicht als Bezeichner verwendet werden dürfen.
  (Groß- & Kleinschreibung beachten!)

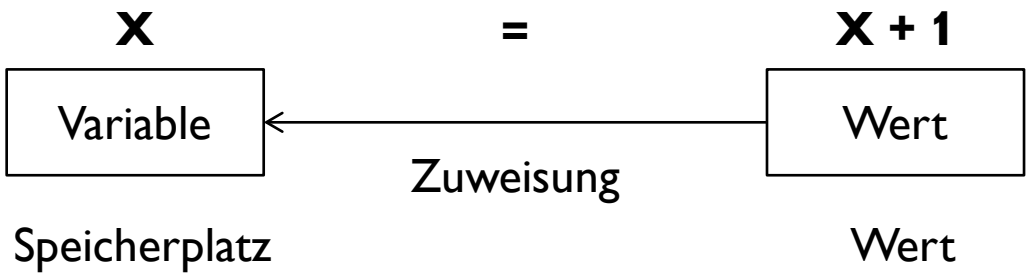
abstract	class	extends	implements	new	static	transient
boolean	const	final	import	package	super	try
break	continue	finally	instanceof	private	switch	void
byte	default	float	int	protected	synchronized	volatile
case	do	for	interface	public	this	while
catch	double	goto	long	return	throw	
char	else	if	native	short	throws	

Namenskonventionen

- ... erhöhen die Lesbarkeit und Wartbarkeit von Programmen.
  Folgende Regeln haben sich etabliert:
  - ⇨ Variablennamen beginnen mit einem Kleinbuchstaben (z.B. args).
    Sie sollten sprechend/sinnvoll gewählt werden!
  - ⇨ Namen von Konstanten bestehen aus Großbuchstaben.
    Einzelne Wörter werden durch _ getrennt.
  - ⇨ Methoden beginnen mit einem Kleinbuchstaben (z.B. `main` oder `print`) und werden i.d.R. nach Verben benannt.
  - ⇨ Klassen- und Interfacenamen beginnen mit einem Großbuchstaben (z.B. `HelloWorld`)
  - ⇨ Namen werden mit gemischten Groß- und Kleinbuchstaben geschrieben, wobei Großbuchstaben zum Trennen von Wortstämmen dienen (z.B. `HelloWorld`).
  - ⇨ Paketnamen bestehen nur aus Kleinbuchstaben.

Variablen und Zuweisungen

- Eine Variable ist ein *Name* für einen *Speicherbereich* für Datenwerte.
- Mit einer Variablen können die folgende elementaren Operationen durchgeführt werden:
  - ⇨ Variable lesen, also den unter dem jeweiligen Namen im Hauptspeicher gespeicherten Datenwert bestimmen.
  - ⇨ Werte einer Variablen zuweisen, also den Speicherbereich der Variablen belegen. Falls dort bereits ein Datenwert abgelegt war, so wird dieser durch die neue Zuweisung überschrieben.
- Eine Zuweisung stellt das einfachste Beispiel einer elementaren Aktion dar. Weitere Beispiele sind etwa die Addition, die zwei ganze Zahlen eine weitere (ihre Summe) zuordnet.



Typisierung von Variablen

- Java zählt zu den streng typisierten Sprachen
  - ⇨ Variablen können nur ganz bestimmte Datenwerte aufnehmen. Dies wird durch den Datentyp festgelegt.
  - ⇨ Der Datentyp bestimmt auch die Operationen, die mit der Variablen ausgeführt werden dürfen.
- In vielen Programmiersprachen (auch in Java) müssen Variablen erst deklariert (vereinbart) werden, bevor sie verwendet werden dürfen.
  - ⇨ Bei der Deklaration muss jeder Variablen ein Typ zugeordnet werden.
  - ⇨ Variablendeklarationen erlauben die Prüfung der Typverträglichkeit von Zuweisungen und anderen Anweisungen bereits zur Compile-Zeit.
  - ⇨ Der Compiler kann bereits den Speicherbedarf der Variablen in Abhängigkeit des Datentyps ermitteln.

Deklaration und Gültigkeit

- Die Position der Variablendeklaration im Programm legt zugleich auch den Gültigkeitsbereich der Variablen fest.
  - ⇨ Variablen können erst nach ihrer Deklaration verwendet werden
  - ⇨ Eine Variable ist nach Verlassen des Blocks, in dem sie vereinbart wurde, nicht mehr gültig.
- Die Deklaration einer Variablen erfolgt in der Form:
  - ⇨ Datentyp *Variablenname;*
- Beispiele:
  - ⇨ int *alter;*
  - ⇨ booleon *allesOk;*

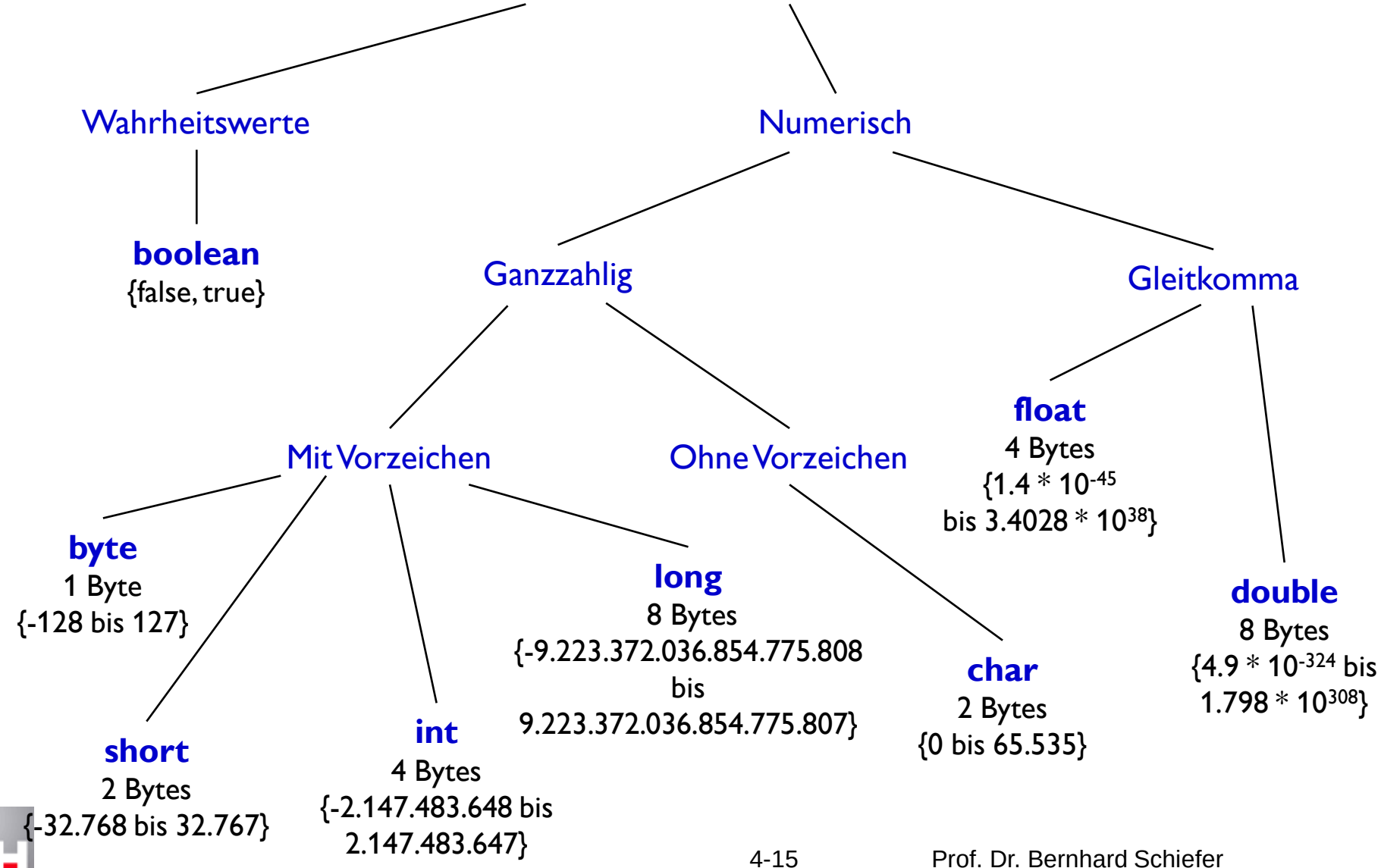
Anweisungen

- Anweisungen stehen neben dem Variablenkonzept ebenfalls im Mittelpunkt imperativer Programmiersprachen. Man unterscheidet:
  - ⇨ **Ausdrücke** (Expression Statement): Durchführen von Berechnungen
  - ⇨ **Deklarationen** (Declaration Statement): Deklaration, Initialisierung (Definition) von Variablen, Konstanten, Datentypen …
  - ⇨ **Kontrollflussanweisung** (Control Flow Statement): zur Steuerung der Logik des Programms (Verzweigungen und Schleifen, …)
- Eine Ausdrucksanweisung wird in Java durch ein Semikolon abgeschlossen.
  - ⇨ Erlaubt sind z.B. : Zuweisungsausdrücke, Inkrement, Dekrement, Methodenaufrufe, …
    - x = x + 2;
    - ++x;
    - System.out.print ("Hello");
    - new BeispielKlasse( ) ;

Datenstrukturen

- Ein Algorithmus beschreibt eine Folge von Operationen, die auf vorgegebenen Daten angewendet werden können.
  - ⇨ Die zulässigen Daten müssen dabei normalerweise bestimmte Eigenschaften haben
  - ⇨ Um mit Hilfe von Variablen, Algorithmen beschreiben zu können, wird der Begriff der „Datenstruktur“ oder als Synonym „Datentyp“ benötigt.
- Eine Datenstruktur legt fest, welche Operationen auf den Daten zulässig sind.
  - ⇨ Eine Operation ist dabei eine Verknüpfung, die einer festen Anzahl von Eingabedaten ein Ausgabedatum zuordnet.
- Eine Programmiersprache enthält eine Reihe elementarer oder sogenannter „build-in“- Datentypen.
  - ⇨ Häufig sind dies von der Maschine direkt unterstützten Datenformate
  - ⇨ In Java werden diese Datentypen als „primitive Typen“ bezeichnet.
- Darauf aufbauend können komplexere benutzerdefinierbare Datentypen definiert werden
  - ⇨ Klassentypen, Interfacetypen, Feldtypen, Enum-Typen, ...

Primitive Datentypen



Wahrheitswerte

- Der logische Typ `boolean` kennt nur genau zwei verschiedene Literale:
  - ⇨ **`true`** und **`false`**.
- Dieser Datentyp wird dort verwendet, wo ein logischer Operand erforderlich ist
  - ⇨ z.B. bei Bedingungen in Fallunterscheidungen und Schleifen.
- Eine direkte Umwandlung der Wahrheitswerte in ganzzahlige Werte ist in Java nicht möglich.
  - ⇨ Manche andere Sprachen (z.B. C, PHP) unterstützen dies
- Der im Speicher tatsächlich belegte Platz für eine boolean-Variable ist nicht definiert und abhängig von der Implementierung der VM

Zeichen

- Der Zeichentyp `char` dient dazu, einzelne Zeichen des Unicode-Zeichensatzes zu speichern.
- Literale werden in einfache Ausführungszeichen eingeschlossen und als Unicode-Zeichen oder als sogenannte Escape-Sequenzen ausgegeben.
  - ⇨ `char zeichen = 'x';`
  - ⇨ `char escapeZeichen = '\"';`
- Liste einiger Escape-Sequenzen:
  - ⇨ \b Backspace (Rücktaste) (Unicode 0008)
  - ⇨ \n Neue Zeile (Newline) (Unicode 000A)
  - ⇨ \r Return (Wagenrücklauf) (Unicode 000D)
  - ⇨ \f Seitenumbruch (Formfeed) (Unicode 000C)
  - ⇨ \u003d Unicode-Zeichen in Hexadezimalschreibweise (Unicode 003d)

Ganze Zahlen

- Die ganzzahligen Typen `byte`, `short`, `int` und `long` sind vorzeichenbehaftet.
- Literale können in Dezimal-, Oktal- oder Hexadezimalform geschrieben werden:
  - ⇨ Ein oktaler Wert beginnt mit dem Präfix 0,
  - ⇨ ein hexadezimaler Wert mit dem Präfix 0x oder 0X.
- Gültige Ziffern sind bei den dezimalen Literalen 0 bis 9, bei oktalen Literalen 0 bis 7 und bei hexadezimalen Literalen a bis f und A bis F.
- Negative Zahlen werden durch Voranstellen des Minuszeichens - dargestellt.
- Ganzzahlige Literale sind vom Typ int, wenn nicht der Buchstabe l oder L angehängt ist. In diesem Fall sind sie vom Typ long.

Gleitkommazahlen

- Literale der Fließkommatypen `float` und `double` werden in Dezimalschreibweise notiert. Sie bestehen aus:
  - ⇨ Vorkommateil (kann ein Vorzeichen + oder - vorangestellt werden)
  - ⇨ Dezimalpunkt
  - ⇨ Nachkommateil
  - ⇨ Exponent e oder E (kann ein Vorzeichen + oder - vorangestellt werden)
  - ⇨ Suffix f oder F für float / d oder D für double (implizit)
- Es **muss mindestens** der **Dezimalpunkt**, der **Exponent oder** das **Suffix** vorhanden sein
  - ⇨ damit eindeutig klar wird, dass es sich um eine Gleitkommazahl handelt
- Entweder Vorkommateil oder der Nachkommateil darf wegfallen.

Zeichenketten

- Zeichenketten sind Folgen von Zeichenliteralen, die in doppelte Anführungszeichen eingeschlossen sind
  - ⇨ Beispiel: "Das ist eine Zeichenkette"
- Einen zugehörigen primitiven Datentyp gibt es allerdings nicht, sondern eine Klasse String, über die Zeichenketten erzeugt werden können.
- Aneinanderreihen von Zeichenketten mit dem Operator +
- Beispiel:

```
class VarTest {
  public static void main (String[ ] args) {
    boolean  info     = true;
    char     zeichen  = 'x';
    int      nummer   = 4711;
    double   wert     = 1.2345;
    System.out.println ("info: " + info);
    System.out.println ("zeichen: " + zeichen);
    System.out.println ("nummer: " + nummer);
    System.out.println ("wert: " + wert);
  } // main
} // class VarTest
```

Konstanten

- Konstanten sind Variablen, die einmalig einen Wert zugewiesen bekommen, welcher dann nicht mehr geändert werden kann.
- In Java werden Konstanten mit Hilfe des Attributs final beschrieben, z.B.:
  - ⇨ `final double PI;` // Noch genau eine weitere Zuweisung möglich
  - ⇨ `final double PI = 3.1415;` // Keine weitere Zuweisung mehr möglich.
- Konstanten werden also dann verwendet, wenn man sicher gehen will, dass bekannte Werte auch immer mit dem intuitiv assoziierten Wert ausgestattet sind.

Vordefiniert Konstanten

- In Java gibt es einige nützliche vordefinierte Konstanten
  - ⇨ z.B. für Randwerte der Gültigkeitsbereiche

```
System.out.println( Byte.MIN_VALUE );       // -128
System.out.println( Byte.MAX_VALUE );       // 127

System.out.println( Character.MIN_VALUE );  // '\u0000'
System.out.println( Character.MAX_VALUE );  // '\uFFFF‘

System.out.println( Integer.MIN_VALUE );  // -2147483648
System.out.println( Integer.MAX_VALUE );  // 2147483647

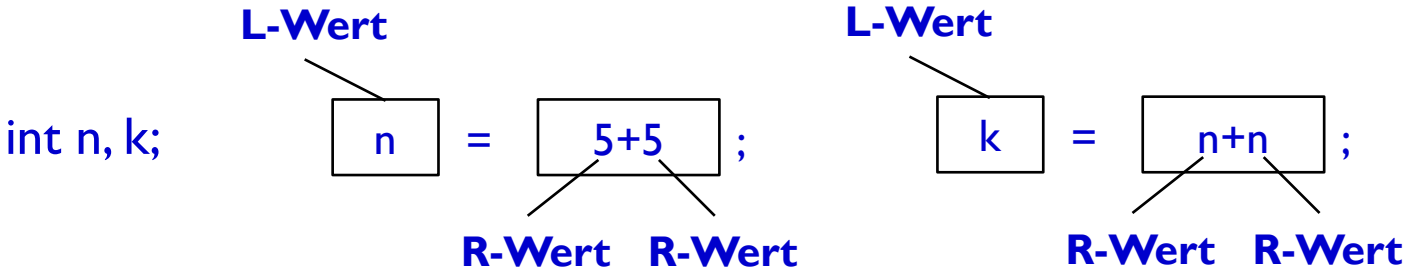
System.out.println( Double.MIN_VALUE );     // 4.9E-324
System.out.println( Double.MAX_VALUE );     // 1.79769313486231570E308
```

Ausdrücke und Operatoren

- Mit **Operatoren** können Zuweisungen und Berechnungen vorgenommen und Bedingungen formuliert und geprüft werden.
- **Ausdrücke** beschreiben eine Auswertungsvorschrift. Sie bestehen aus Operanden, auf die ein Operator angewandt wird (evtl. auch Klammern).
  - ⇨ Jeder Ausdruck hat einen **Rückgabewert**, dessen Typ sich aus den Typen der Operanden und der Art des Operators bestimmt.
  - ⇨ **Operanden** können Variable und Literale, aber auch Methodenaufrufe (Prozeduraufrufe) sein.
  - ⇨ Wenn mehrere Operatoren im Ausdruck vorkommen, wird die Reihenfolge der Auswertung durch Vorrangregeln (Prioritäten) festgelegt. Durch Setzen von runden Klammern lässt sich eine bestimmte Auswertungsreihenfolge erzwingen.
- Man unterscheidet arithmetische, relationale, logische, Bit-, Zuweisungs- und sonstige Operatoren.

L-Werte und R-Werte

- Einen Ausdruck, der eine Variable im Speicher bezeichnet, nennt man einen **L-Wert** (lvalue oder left value).
  - ⇨ Ein L-Wert besitzt einen Speicherplatz im Arbeitsspeicher.
- Steht ein Variablennamen rechts neben dem Zuweisungsoperator, so wird über den Variablennamen der Wert an der entsprechenden Speicherstelle ausgelesen. Es interessiert hier nur sein **R-Wert** (rvalue oder right value).
- Links neben dem Zuweisungsoperator muss immer ein L-Wert stehen, da man eine Speicherstelle benötigt, die den Wert der Zuweisung aufnehmen kann.
- Ist ein Ausdruck kein L-Wert, so ist er ein R-Wert und kann nur rechts vom Zuweisungsoperator stehen. Einem R-Wert kann kein Wert zugewiesen werden, da er keine feste Speicherstelle besitzt.



Arithmetische Operatoren

- Die arithmetischen Operatoren haben numerische Operanden und liefern einen numerischen Rückgabewert.
- Haben die Operanden unterschiedliche Datentypen, so wird automatisch (implizit) eine Typumwandlung „nach oben" durchgeführt.
  - ⇨ Der kleinere Typ der beiden Operanden wird in den Typ des größeren umgewandelt.
  - ⇨ Der Ergebnistyp entspricht dem größeren der beiden Operanden.
- Implizite Typumwandlung:
  - ⇨ Numerische Typen: byte → short → int → long → float → double
  - ⇨ Zeichen: char → int
- Explizite Typumwandlung: Bei der Umwandlung von groß → kleinerem Datentyp wird nur eine explizite Typumwandlung erlaubt da Genauigkeitsverlust. Dafür gibt es den Cast-Operator

Arithmetische Operatoren

- Der Modulo-Operator % berechnet den Rest einer Division, der wie üblich definiert ist:
  - ⇨ Rest = Zähler % Nenner = Zähler - *Anzahl* * Nenner,
  - ⇨ wobei *Anzahl* eine ganze Zahl und Rest < Nenner ist.
- Beim einstelligen Inkrement- (++) und Dekrementoperator (--), der sich nur auf Variablen anwenden lässt, wird zwischen Präfix- und Postfixform unterschieden, je nach- dem, ob der Operator vor oder hinter dem Operanden steht:
  - ⇨ ++var, --var; (Wert wird vor der Nutzung verändert)
  - ⇨ var++, var--; (Wert wird nach der Nutzung verändert)

Nebeneffekte

- Die einstelligen Operatoren erlauben eine schnelle und kurze Programmschreibweise, besitzen dafür aber Nebeneffekte (auch Seiteneffekte oder Nebenwirkungen genannt)
- Dazu schauen wir uns das folgende Beispiel an:
  - ```
    int n = 1;
    int m;
    m = n++;
    ```
  - Der Rückgabewert des Ausdrucks n++ ist hier der Wert 1.
  - Mit dem Zuweisungsoperator wird der Variablen m ebenfalls der Wert 1 zugewiesen.
  - Die Zuweisung m = n++ ist ebenfalls ein Ausdruck und m = n++; stellt eine Ausdrucksanweisung dar. Als Nebeneffekt des Operators ++ wird die Variable n inkrementiert und hat danach den Wert 2.
    ```

Nebeneffekte

- Grundsätzlich sollte man mit Nebeneffekten sparsam umgehen, da sie leicht zu unleserlichen und fehlerträchtigen Programmen führen.
- In Java gibt es zwei Sorten von Nebeneffekten:
  - ⇨ Nebeneffekte von Operatoren (siehe Beispiel) und
  - ⇨ Nebeneffekte bei allen Methoden, die nicht nur lesend sondern auch schreibend auf Datenfelder zugreifen (siehe später: Kapitel - Objekte)

Arithmetische Operatoren: Rangfolge

Operator	Operator	Priorität
+	positives Vorzeichen	1
-	negatives Vorzeichen	1
++	Inkrement	1
--	Dekrement	1
*	Multiplikation	2
/	Division	2
%	Modulo	2
+	Addition	3
-	Subtraktion	3

Beispiel

```
class ArithmOp {
   public static void main (String[ ] args)  {

      double  val1 = 13.8, val2 = 5.6;
      int     number = 5;

      System.out.println ("val1 / val2: "  +  val1 / val2);
      System.out.println ("val1 % val2: "  +  val1 % val2);

      System.out.print ("++number: "  +  ++number + " , ");
      System.out.println ("number: "  +  number);
      System.out.print ("number++: "  +  number++ + " , ");
      System.out.println ("number: "  +  number);

      System.out.print ("--number: "  +  --number + " , ");
      System.out.println ("number: "  +  number);
      System.out.print ("number--: "  +  number-- + " , ");
      System.out.println ("number: "  +  number);
   } // main
} // class ArithmOp
```

Relationale Operatoren

- Relationale Operatoren vergleichen Ausdrücke mit numerischen Werten miteinander. Das Ergebnis ist vom Typ boolean:

Operator	Bezeichnung	Priorität
<	kleiner	5
<=	kleiner oder gleich	5
>	größer	5
>=	größer oder gleich	5
==	gleich	6
!=	ungleich	6

- Bei Gleitkommawerten sollte die Prüfung auf **exakte** Gleichheit oder Ungleichheit vermieden werden, da es bei aufwendigen Berechnungen zu Rundungsfehlern kommen kann.
- Statt dessen?

Logische Operatoren

- Logische Operatoren verknüpfen Wahrheitswerte miteinander.
  - ⇨ Java stellt die Operationen UND, ODER, NICHT und das exklusive ODER zur Verfügung:

Operator	Bezeichnung	Priorität
!	NICHT	1
&	UND mit vollst. Auswertung	7
^	exklusives ODER	8
|	ODER mit vollst. Auswertung	9
&&	UND mit kurzer Auswertung	10
||	ODER mit kurzer Auswertung	11

- Man sieht: UND und ODER gibt es in zwei Varianten.
  - ⇨ Bei der sogenannten kurzen Variante wird der zweite Operand nicht mehr ausgewertet, wenn das Gesamtergebnis bereits feststeht.
  - ⇨ Beispiele dafür?

Logische Operatoren

- Wahrheitstabellen für alle Operatoren:

a	b	a & b a && b	a ^ b	a | b a || b
true	true	true	false	true
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

a	! a
true	false
false	true

Beispiel

```
class LogOp {

  public static void main (String[ ] args)  {
     int num1 = 2, num2 = 3;

     System.out.println (num1 == 2  && num2 < 8);
     System.out.println (num1 != 2  || !(num2 < 2));
     System.out.println (num1 == 2  ^ num2 > 0);

    } // main
 } // class LogOp
```

Bit-Operatoren

- Bit-Operatoren arbeiten auf der Binärdarstellung der numerischen Operanden, also mit 8 (byte), 16 (short), 32 (int) oder 64 Bit (long):

Operator	Bezeichnung	Priorität
~	Bit-Komplement	1
<<	Links-Schieben (Vorzeichen behaftet)	4
>>	Rechts-Schieben (Vorzeichen behaftet)	4
>>>	Rechts-Schieben m. Nachziehen von Nullen	4
&	Bitweises UND	7
^	Bitweises exklusives ODER	8
|	Bitweises ODER	9

- ~var entsteht aus var, indem alle Bit von var invertiert werden (0 geht in 1 über und umgekehrt).
- Bei den Schiebeoperationen werden alle Bit des ersten Operanden um so viele Stellen nach links/rechts geschoben, wie im zweiten Operanden angegeben ist.

Schiebe-Operatoren

- Bei Beispielen wird aus Gründen der Einfachheit ein **Halbbyte** verwendet
- **Links-Schieben**: (stellt **arithmetische Operation** dar)
  - ⇨ int n; n << k entspricht (bis auf Extreme am Rand) der Multiplikation mit 2^k, also n * 2^k.
  - ⇨ **1 << 2** = 1 * 2^2 = **4**: 0001 **<< 2** ==> 01**00**
  - ⇨ **- 4 << 1** = -4 * 2^1 = **- 8**: 1100 **<< 1** ==> 100**0**
- **Rechts-Schieben** (stellt **arithmetische Operation** dar)
  - ⇨ int n; n >> k entspricht (bis auf Extreme am Rand) der Divison mit 2^k, also n / 2^k.
  - ⇨ **4 >> 2** = 4 / 2^2 = **1**: 0100 **>> 2** ==> **00**01
  - ⇨ **- 8 >> 2** = - 8 / 2^2= **- 2**: 1000 **>> 2** ==> **11**10
- Beim Operator >>> werden von links immer Nullen nachgezogen

Zuweisungs-Operatoren

- Neben der einfachen Zuweisung können arithmetische und bitweise Operatoren mit der Zuweisung kombiniert werden:

Operator	Bezeichnung	Priorität
=	Einfache Zuweisung	13
op=	kombinierte Zuweisung	13

Dabei steht *op* für *, /, %, +, -, <<, >>, >>>, &, ^ und | .

- Bei der einfachen Zuweisung = wird der rechts stehende Ausdruck ausgewertet und der links stehenden Variablen zugewiesen.
  - ⇨ Dabei müssen die Datentypen beider Seiten kompatibel sein.
  - ⇨ Falls nötig, wird eine implizite Typanpassung (Umwandlung) der rechten Seite nach „oben“ in den Typ der Variablen durchgeführt.
- a *op*= b entspricht der Zuweisung: a = a *op* b, wobei für *op* der anzuwendende Operator einzusetzen ist, z.B. a += b: a = a + b.

Beispiel

- Beachten:
  - ⇨ Mehrfachzuweisungen sind rechts-assoziativ
  - ⇨ Zuweisungsoperator hat die geringste Priorität

```
class ZuweisOp {
   public static void main (String[ ] args)  {

      int num1, num2 = 5, num3 = 9;

      num1 = num2 + num3;
      System.out.println ("num1: " + num1);
      num1 += num2;
      System.out.println ("num1: " + num1);
      num1 = num2 = 0;
      System.out.println ("num1: " + num1 + " , num2: " + num2);

   } // main
} // class ZuweisOp
```

Bedingungs-Operator

- Der Bedingungs-Operator **? :** besitzt drei Operanden:
  - ⇨ Bedingung ? Ausdruck1 (true) : Ausdruck 2 (false)
  - ⇨ Die Bedingung stellt einen logischen Ausdruck dar.
  - ⇨ Falls Bedingung den Wahrheitswert true liefert, ergibt sich der Wert von Ausdruck 1,
  - ⇨ sonst der Wert von Ausdruck2.
  - ⇨ Der Bedingungsoperator hat die Priorität 12.
- Beispiele:
  - ⇨ `x = a < 0 ? -a : a`
  - ⇨ `y = a < b ? b : a`

Cast-Operator

- Mit dem Cast-Operator wird eine explizite Typumwandlung vorgenommen:
  - ⇨ Der Ausdruck **(type) a** wandelt den Ausdruck vom Typ type um.
  - ⇨ Der Cast-Operator darf nur auf der rechten Seite einer Zuweisung stehen.
  - ⇨ Der Cast-Operator hat die Priorität 1.
- Der Programmierer muss sich genauestens bewusst sein, was er tut, da es bei einer expliziten Typumwandlung zu einem Genauigkeitsverlust kommt, wodurch auch wichtige Daten verloren gehen können oder es zu Fehlern kommen kann.

Cast-Operator

- Beispiel:

```
class CastOp {
   public static void main (String[ ] args)  {

      double x;
      int n = 5, m =3;

      x = n / m;
      System.out.println ("x: " + x) ;
      x = (double) n / m;
      System.out.println ("x: " + x) ;

   } // main
} // class CastOp
```

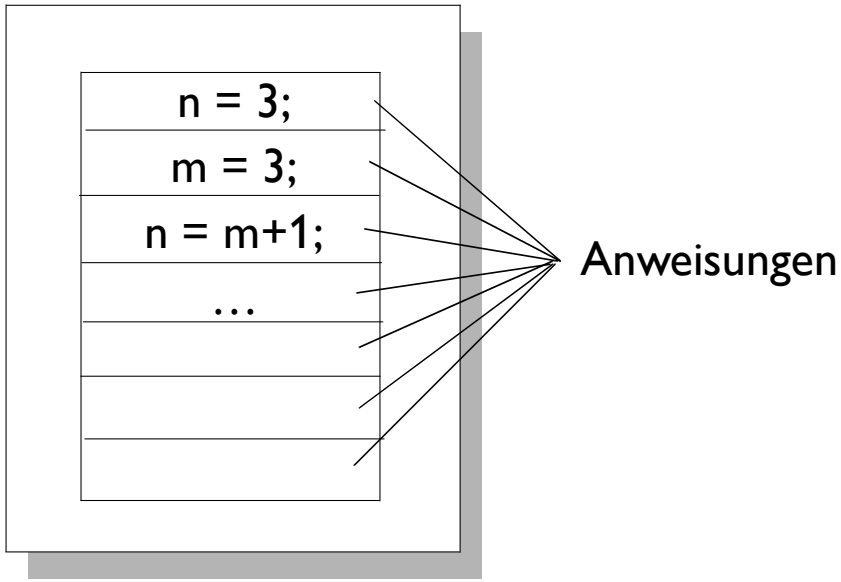
Sequentielle Komposition – Blöcke

- Die sequentielle Komposition bezeichnet die sukzessive Ausführung von Anweisungen.
- Ein Block wird syntaktisch als Paar von geschweiften Klammern, die Anweisungen enthalten, definiert:
  - ⇨ Die einzelnen Anweisungen werden in einem Block zusammengefasst und dann hintereinander ausgeführt.
  - ⇨ Formal wird der gesamte Anweisungsblock wie eine einzige Anweisung interpretiert und kann überall dort verwendet werden, wo eine elementare Anweisung erlaubt ist.
- Blöcke können ineinander geschachtelt werden.
- Variablen, die in einem Block deklariert werden, sind nur dort gültig und sichtbar.

Sequentielle Komposition

```
{
   Anweisung_1
   Anweisung_2
   . . .
   Anweisung_n

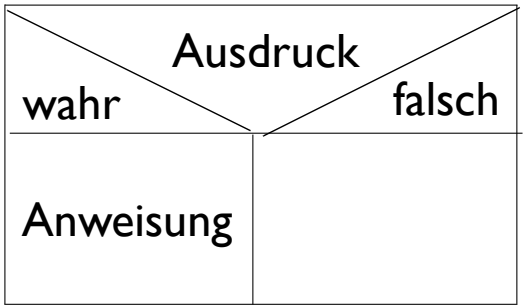
} // block Beispiel
```



Auswahl / Entscheidung / Verzweigung

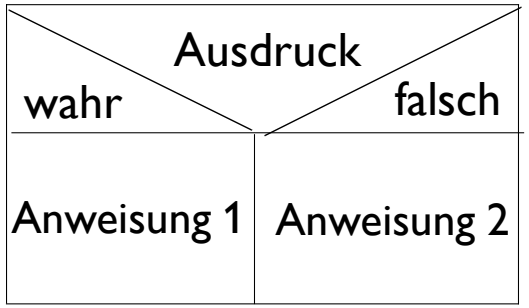
- Soll ein Block sequentieller Anweisungen nicht immer, sondern nur unter einer speziellen Bedingung ausgeführt werden, wird ein Entscheidungskonstrukt der folgenden Art benötigt:
  - ⇨ Wenn (Bedingung) dann <Anweisung> sonst <andere Anweisung>
- In Java gibt es zwei Varianten dieser Art (bedingte Ausführungen):
  - ⇨ Bedingte Anweisung:

```
if (Ausdruck)
   Anweisung
```



  - ⇨ Alternative:

```
if (Ausdruck)
   Anweisung 1
else
   Anweisung 2
```



- Der else-Block ist also optional

Beispiel

```
class IfTest {
    public static void main (String[ ] args)  {
       int n = 2, m, k = -1;

      if (n >= 0)  {
         System.out.println ("n ist groesser gleich 0");
         m = -1;
      } else {
         System.out.println ("n ist kleiner 0");
         m = 1;
      } // else

      if (m == 1)
         if (k  > 0)
            System.out.println ("k ist groesser 0");
         else
            System.out.println ("k ist kleiner gleich 0");
    } // main
} // class IfTest
```

Mehrfache Alternative – else-if

- Die else-if Kombination ist die allgemeinste Möglichkeit einer Mehrfach-Selektion (treffen einer Auswahl unter verschiedenen Alternativen):

  ⇨
```
if (Ausdruck_1)
    Anweisung_1
else if (Ausdruck_2)
    Anweisung_2
    ...
else Anweisung_n
```

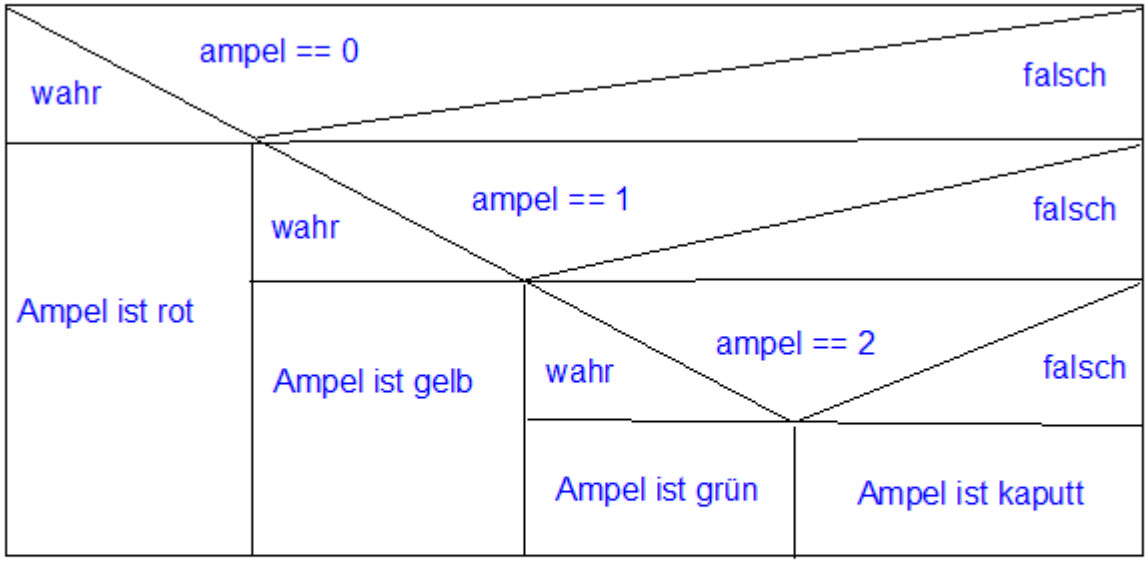
- In der angegebenen Reihenfolge wird ein Vergleich nach dem anderen durchgeführt.

  ⇨ Bei der ersten Bedingung, die erfüllt ist, wird die zugehörige Anweisung abgearbeitet und die Mehrfach-Selektion abgebrochen.

  ⇨ Der Ausdruck kann in jedem if völlig anders aufgebaut sein

- Beachten: Eine Anweisung kann stets auch ein Block sein.

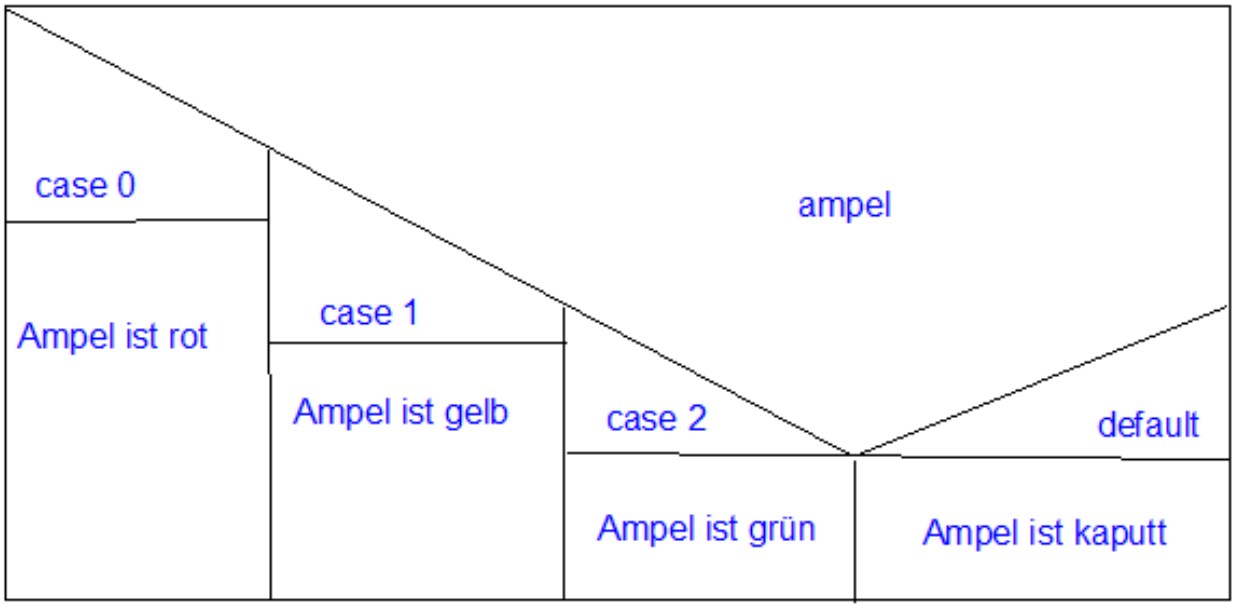
Mehrfach-Alternative: Ampel-Beispiel



Switch-Anweisung

- Wenn die Verzweigung anhand der verschiedenen Werte des immer gleichen Ausdrucks geschehen soll, gibt es bessere Alternativen
  - ⇨ In Java (und vielen anderen Sprachen) gibt es für diesen Fall sogenannte Schalter oder switch-Anweisungen.
- Die **`switch`**-Anweisung führt je nach Wert des Ausdrucks, den sie wie ein Argument erhält, alternative Anweisungen aus.
- Der Einstieg erfolgt bei der ersten passenden **`case`**-Marke
- Anschließend werden alle weiteren Anweisungen ausgeführt
  - ⇨ Um die `switch`-Anweisung vorzeitig zu verlassen, ist ein **`break;`** erforderlich.
- Falls kein passender Wert mit `case` definiert wurde, werden – falls vorhanden – die Anweisungen des optionalen **`default`** Falls ausgeführt.

Switch-Anweisung: Ampel-Beispiel



Beispiel

- Java Switch-Anweisung :

```
class SwitchTest {
  public static void main (String[ ] args)  {

    char auswahl = 's';

    switch (auswahl)  {
      case 'n':
      case 'N': System.out.println ("Option Neu");
                break;
      case 'o':
      case 'O': System.out.println ("Option Öffnen");
                break;
      case 's':
      case 'S': System.out.println ("Option Speichern");
                break;
      default:  System.out.println ("Option ungültig");
    } // switch
  } // main
} // class SwitchTest
```

Schleifen

- Schleifen werden benötigt, wenn Anweisungen *wiederholt* ausgeführt werden sollen
  - ⇨ wie z.B. in Kapitel „Grundlagen der Programmierung“: Flussdiagramm zur Ermittlung des ggT.
- Schleifen führen Anweisungen wiederholt aus, solange bis eine Bedingung erfüllt ist
  - ⇨ eine andere Möglichkeit einer Schleife ist eine Iteration z.B. durch eine Liste.
- In Java werden im wesentlichen drei Schleifen-Varianten angeboten, die im Folgenden vorgestellt werden.
- Viele Fehlerquellen beim Programmieren passieren bei inkorrekt formulierten Schleifenbedingungen, sodass Endlosschleifen entstehen können, welche jedoch meist unerwünscht sind.

while-Schleife

- Die **while-Schleife** ist eine **abweisende** Schleife, d.h. die (Ausführungs-) Bedingung wird jeweils *vor* Eintritt in die Schleife überprüft:
  - ⇨ **while** (Ausdruck) Anweisung
- Der Ausdruck muss vom (Ergebnis-)Typ `boolean` sein
- Wenn der Ausdruck den Wert true hat, wird die Blockanweisung ausgeführt und bei Erreichen des Blockendes wieder zum Ausdruck gesprungen, um diesen neu zu prüfen.
- Wird der Ausdruck mit false ausgewertet, wird mit der Anweisung fortgefahren, die der Anweisung (dem Schleifenkörper) folgt.

do-while-Schleife

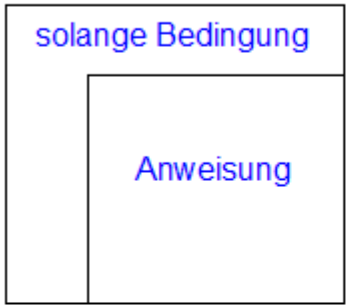
- Die **do-while-Schleife** ist eine **annehmende** (nicht abweisende) Schleife, d.h. die Anweisungen im Schleifenkörper werden mindestens einmal ausgeführt.
  - ⇨ **do** Anweisung **while** (Ausdruck)
- Funktioniert ähnlich wie die while-Schleife, allerdings wird hier der Anweisungsblock mindestens einmal (do) ausgeführt, erst danach wird der Ausdruck auf true oder false überprüft (while).
- Die Schleife wird beendet, wenn der Ausdruck den Wert false hat.

for-Schleife

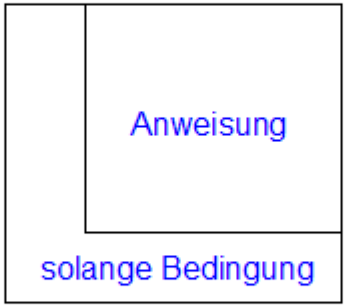
- Die **for-Schleife** ist eine **aufzählende** Schleife und wiederholt eine Anweisung in Abhängigkeit von Kontrollausdrücken.
  - ⇨ for (Init; Bedingung; Update) Schleifenanweisung
- *Init* wird bei Beginn der Schleife abgearbeitet und ist eine Ausdrucksanweisung (ohne Semikolon) oder eine durch Kommata getrennte Folge von Anweisungen. Darf auch fehlen.
- *Bedingung* ist ein Ausdruck vom Typ boolean; wird zum Beginn jedes Schleifendurchgangs getestet und die Schleifenanweisungen werden nur durchgeführt, wenn die Bedingung true ist.
- *Update* wird nach jedem Schleifendurchgang durchgeführt; ist eine Anweisung oder mit Kommata getrennte Folge von Anweisungen.

Schleifen-Struktogramme

Abweisende Schleife

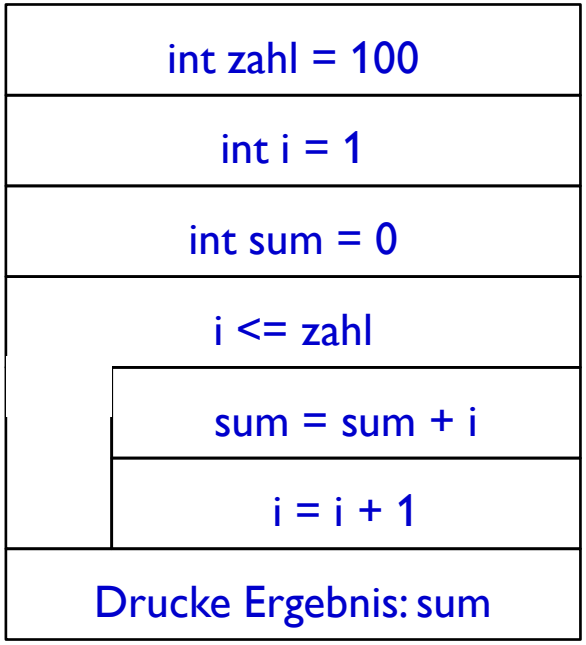


Annehmende Schleife



Aufzählende Schleife

Beispiel: Summe (Abweisende Schleife)



Beispiel: Summe (while-Konstrukt)

```
class sumWhileTest {
    public static void main (String[ ] args)  {
       int zahl = 100;  // Bis zu dieser Zahl summieren.
       int i = 1;       // Laufvariable für while-Scheife
       int sum = 0;     // Nimmt die (temporären) Summen auf

       while (i <= zahl)  {
          sum  += i;
          i++;
       } // while
       System.out.println (" Summe 1 bis " + zahl + ":" + sum);
   } // main
} // class sumWhileTest
```

Beispiel: ggT in mathematischer Notation

- Der entsprechende Algorithmus wurde bereits mathematisch wie folgt rekursiv beschrieben:

$$ggT(a,b) = \begin{cases} a & \text{falls } b = 0 \\ ggT(b,a) & \text{falls } b > a \\ ggT(b,a \bmod b) & \text{sonst} \end{cases}$$

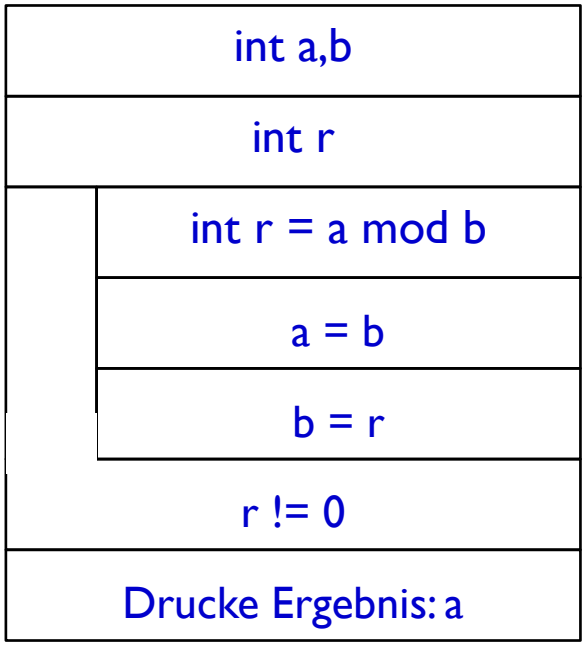
  - ⇨ Bitte beachten, dass a ≥ 0 und b > 0 gilt, so dass der Fall b=0 nicht zu Beginn gelten kann, sondern erst innerhalb einer echten Rekursionstiefe.

- Falls b > a gilt, folgt sofort die folgende Aussage a mod b = a.
  - ⇨ Damit folgt insgesamt: ggT (b,a) = ggT (b, a mod b).

- Daher kann der ggT vereinfacht ausgedrückt werden und zwar durch Zurückführen der Darstellung der 2. Alternative auf die 3. Alternative:

$$ggT(a,b) = \begin{cases} a & \text{falls } b = 0 \\ ggT(b,a \bmod b) & \text{sonst} \end{cases}$$

Beispiel: ggT in Nassi-Shneidermann - Notation



Nimmt Eingabewerte auf

Nimmt Restwert auf

Beispiel: ggT (do-while-Konstrukt)

```
class ggTwhileTest {
   public static void main (String[ ] args)  {
      int a, b, r;
      a = 960;
      b = 230;

      do {
         r = a % b;
         a = b;
         b = r;
      } while (r != 0);

      System.out.println (" ggT = " + a);

   } // main
} // class ggTwhileTest
```

Beispiel: ggT (while-Konstrukt)

```
class ggTwhileTest {
  public static void main (String[ ] args)  {

        int a, b, r;
        a = 960;
        b = 230;
        r = a % b;

        while (r != 0) {
           a = b;
           b = r;
           r = a % b;
        } // while

        System.out.println (" ggt = " + b);

  } // main
} // class ggTwhileTest
```

Beispiel: Summe (for-Konstrukt)

```
class sumForTest {
  public static void main (String[ ] args)  {

    int zahl = 100; // Bis zu dieser Zahl soll die Summe
                    // durchgeführt werden.
    int sum = 0;    // Nimmt die (temporären) Summen auf

    for (int i =1; i <= zahl; ++i)  {
      sum  += i;
    } // for

    System.out.println (" Summe 1 bis " + zahl + ":" + sum);

  } // main
} // class sumForTest
```

Endlos-Schleifen

- Manchmal werden Endlos-Schleifen benötigt.
  - ⇨ Beispiel: Ein Web-Server, der so lange auf Anfragen warten soll, bis er irgendwann abgeschossen wird.
- Wie werden Endlos-Schleifen in Java realisiert?
  - ⇨ Falls der Boolsche Ausdruck in der for-Schleife fehlt, so gilt die Bedingung immer als erfüllt und die Schleife wird automatisch nicht mehr beendet. Häufig werden dann alle drei Ausdrücke weggelassen:
    - `for ( ; ; ) { . . . }`
  - ⇨ Eine elegantere Form ist die while-Schleife, wobei die Bedingung auf die Konstrante `true` gesetzt wird:
    - `while (true) { . . . }`

Sprunganweisungen

- Sprunganweisungen werden verwendet, um Schleifendurchgänge vorzeitig zu beenden. Man unterscheidet:
  - ⇨ Die Anweisung **break** beendet eine while-, do-, for- oder switch-Anweisung, welche die break-Anweisung unmittelbar umgibt.
  - ⇨ Um aus geschachtelten Schleifen herauszuspringen, gibt es die Variante: **break *Marke***
    - Marke steht für einen selbst gewählten Bezeichner.
      Diese Anweisung springt an das Ende der Anweisung, an deren Anfang diese Marke steht.
  - ⇨ Eine markierte Anweisung hat die Form: ***Marke:* Anweisung**
  - ⇨ Die Anweisung **`continue`** unterbricht den aktuellen Schleifendurchgang einer while, do- und for-Schleife und springt an die Wiederholungsbedingung der sie unmittelbar umgebenden Schleife.
  - ⇨ Ähnlich wie bei der break-Anweisung gibt es auch hier die Variante mit Marke: **`continue`** ***Marke***

Beispiel: break

- Was wird hier ausgegeben?

```
class switchBreakTest {
   public static void main (String[ ] args)  {

      int month = 3;

      switch (month) {
         case 1: System.out.println ("Monat Januar");
         case 2: System.out.println ("Monat Februar");
         case 3: System.out.println ("Monat März");
         case 4: System.out.println ("Monat April");
         case 5: System.out.println ("Monat Mai");
         case 6: System.out.println ("Monat Juni") ;
      } // switch
   } // main
} // class switchBreakTest
```

Unterschied: continue vs. break

- Das folgende Beispiel soll den Unterschied zwischen den beiden Abbruchanweisungen verdeutlichen:

```
class ContinueBreakTest {
   public static void main (String[ ] args) {

      for (int i = 1; i <= 10; ++i) {
         if ( i == 5)
            break;  // oder continue
        System.out.println (i);
     } // for

   } // main
} // class ContinueBreakTest
```

- Was wird jeweils ausgegeben?

Beispiel: Sprunganweisungen mit Marke

```
class SprungTest {
  public static void main (String[ ] args) {

    int i;
    Hauptschleife:
    for (i = 1; i <= 10; ++i) { // Hauptschleife
       for (int j = 1; j <= 10; ++j) {
          System.out.print (i * j + "\t");
          if ( j == 5 ) {
             System.out.println ( );
             continue Hauptschleife;
          } // if
       } // for j
    } // for i

    System.out.println ("Wert von i: " + i); // Das geht!
    System.out.println ("Wert von j: " + j);  // Das funktioniert nicht!
  } // main
} // class SprungTest
```

Der Sprungbefehl goto

- Manche Programmiersprachen stellen einen absoluten Sprungbefehl zur Verfügung, meistens mit dem Befehlsnamen **goto (Marke)**:
  - ⇨ Dieser erzwingt ein Verlassen des aktuellen Programmablaufs und ein Sprung zu einer beliebigen, vorher festgelegten Stelle (Marke) des Programmcodes.
  - ⇨ Programme werden durch diese Anweisung sehr undurchsichtig und somit fehleranfällig sowie schwer modifizierbar.
- Aus diesen Gründen verzichten viele moderne Sprachen, auch Java, auf diese Anweisung.
- Aber: Auch continue und break sind in der Kombination mit Marken nicht unproblematisch und führen schnell zu Spaghetti-Code

Beispiel: Fakultät

- Programmier-Aufgabe:
  - ⇨ Implementieren Sie ein Java-Programm, welches zu einer vorgegebenen natürlichen Zahl n (z.B. n = 10), die Fakultät n! berechnet.
  - ⇨ Die Fakultät von n ist bekanntermaßen wie folgt definiert:
    n! = 1 * 2 * . . . * (n-1) * n