Relationale Operatoren

- vergleichen numerische Ausdrücke miteinander
- Ergebnisse haben den Datentyp `boolean`

Operator	Bezeichnung	Beispiel	Wert
`<`	kleiner	`3 < 3`	`false`
`<=`	kleiner-gleich	`3 <= 3`	`true`
`>`	größer	`3.1 > 3.0`	`true`
`>=`	größer-gleich	`-2 >= 0`	`false`
`==`	gleich	`3 == 4`	`false`
`!=`	ungleich	`3 != 4`	`true`

Achtung: Gleichheits-Test von Fließkommazahlen riskant

Logische Operatoren

verknüpfen Wahrheitswerte miteinander

Operator	Bezeichnung
!	nicht
&&	und
||	oder
^	exklusives oder

Achtung: „`&&`“ und „`||`“ werten den 2. Operanden ggfs. nicht aus

- „`!a`“ ist
  - ★ `true`, wenn `a` den Wert `false` besitzt
  - ★ `false`, wenn `a` den Wert `true` besitzt

Wahrheitstabellen

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

Hinweis:

Logische Operatoren können kombiniert und mit runden Klammern „ (...) “ strukturiert werden

Typumwandlung

`(type)Ausdruck`

- wandelt den *Ausdruck* in einen Ausdruck vom Datentyp *type* um
- Umwandlung von Fließkomma- in Integer-Datentypen: Nachkommastellen werden abgeschnitten
- Informationsverlust bei Umwandlung in Datentyp mit niedriger Kapazität
  Beispiel:

```
(int) 4/3.0
```

Ablaufsteuerung

Verzweigungen I

- `if(`*`Ausdruck`*`)` *`Anweisung`*
  führt die *`Anweisung`* aus, wenn der Boolsche *`Ausdruck`* den Wert `true` hat
- `if(`*`Ausdruck`*`)` *`Anweisung1`* `else` *`Anweisung2`*
  führt

  ★ *`Anweisung1`* aus, wenn der Boolsche *`Ausdruck`* den Wert `true` hat

  ★ *`Anweisung2`* aus, wenn der Boolsche *`Ausdruck`* den Wert `false` hat

Verzweigungen I

Beispiel:

```
// Datei: Beispiel2.java

public class Beispiel2 {
      public static void main(String[] args) {
      double a = 3.2;
      if((a >= 2) && (a <= 4))
        {
         System.out.print("a = " + a);
         System.out.println(" liegt zwischen 2 und 4");
         }
      else if(a < 0)
         System.out.println("a ist negativ");
      else
         System.out.println("0<=a<2 oder a>4");
   }
}
```

Verzweigungen 2

- ```
  switch(Ausdruck) {
    case Konstante:
      Anweisungen1
     ...
    default:
      Anweisungen2
  }
  ```

- führt

  ★ *alle* Anweisungen ab *`Anweisungen1`* (ggfs. bis zu einer „`break;`“-Anweisung) aus, wenn der ganzzahlige *`Ausdruck`* den Wert *`Konstante`* hat

  ★ *`Anweisungen2`* aus, wenn der ganzzahlige *`Ausdruck`* keinen der angegebenen Werte hat
  ```

Verzweigungen 2

Beispiel:

```
// Datei: Beispiel3.java

public class Beispiel3 {
       public static void main(String[] args) {
       int a = 6;
       switch(a)
         {
          case 6:
           System.out.println("Glueckwunsch: Sechs");
           break;
          case 1:
           System.out.println("Schade, nur eine Eins");
           break;
          default:
           System.out.println("nichts besonderes");
         }
    }
}
```

Schleifen I

- `while(`*`Ausdruck`*`)`
  `Anweisung`
  führt die *`Anweisung`* solange aus, wie der Boolsche *`Ausdruck`* den Wert `true` hat

  - die Bedingung wird **vor** der Schleife geprüft

Beispiel:

```
int i=1, summe=0;
while(i<=10)
  summe += i++;
System.out.println("Summe = " + summe);
```

  - summiert die ganzen Zahlen 1, ..., 10 auf

Schleifen 2

- `do`

  `Anweisung`

  `while(`*`Ausdruck`*`);`

  führt die *`Anweisung`* solange aus, wie der Boolsche *`Ausdruck`* den Wert `true` hat

- die Bedingung wird erst **am Ende** der Schleife geprüft

- somit wird die *`Anweisung`* mindestens einmal ausgeführt

Schleifen 2

Beispiel:

```
int a=-17;
double zahl=0;

do {
 a--;
 zahl += 0.2*a;
 } while(a > 0);

System.out.println("a = " + a);
System.out.println("zahl = " + zahl);
```

- die Schleife wird einmal durchlaufen, obwohl die Abbruchbedingung bereits beim ersten Durchlauf nicht erfüllt ist

Schleifen 3

- `for(`*`Init`*`; `*`Bedingung`*`; `*`Update`*`)`
  `Anweisung`
- *`Init`*: eine oder mehrere von Kommas getrennte Initalisierungsanweisungen
- *`Bedingung`*: Abbruchbedingung – wird zu Beginn jeden Schleifendurchlaufs geprüft
- *`Update`*: eine oder mehrere von Kommas getrennte Aktualisierungssanweisungen vor dem nächsten Schleifendurchlauf

Schleifen 3

Beispiel:

```
double summe=0;

int b=3;
for(int a=17; a > b; a--, b += 2)
  summe += a/(double) b;

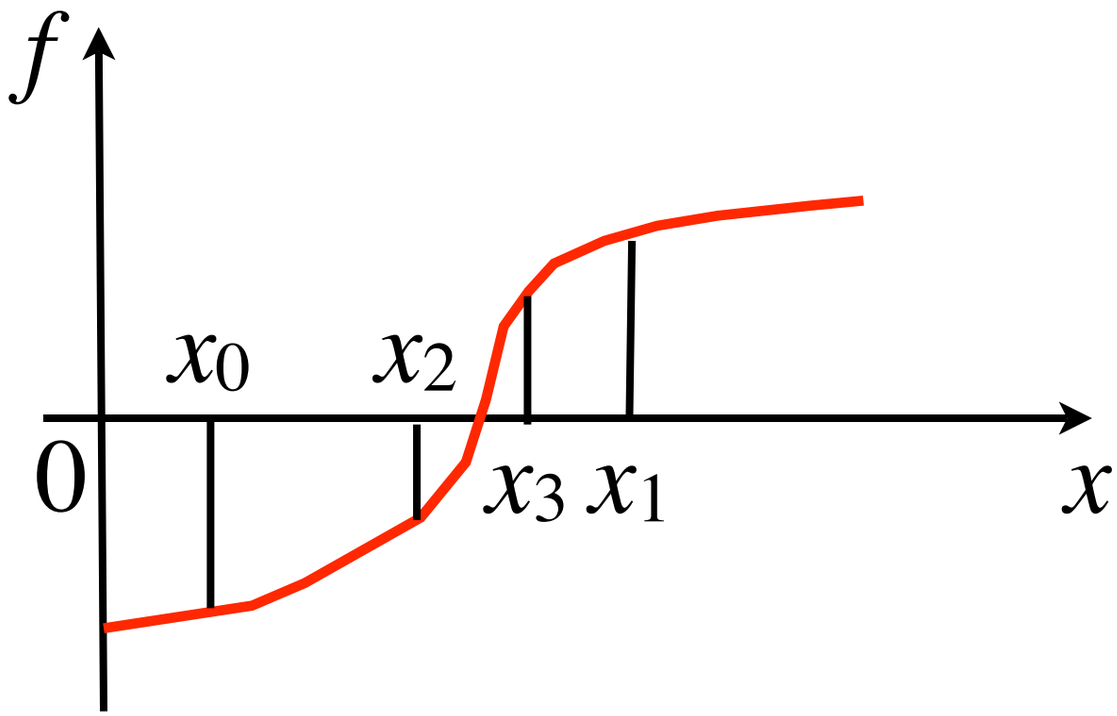
System.out.println("summe = " + summe);
```

- die Variable „a“ wird zu Beginn der Schleife deklariert und initialisiert
- die Schleife enthält zwei Aktualisierungs-sanweisungen

Einschub:

Nullstellen: Bisektionsverfahren

gesucht: Lösung x von $f(x)=0$



- Annahme:
  $f(x_0)<0, f(x_1)>0$
- $x_2 = (x_0 + x_1)/2$
- Hat $f(x_0)$ oder $f(x_1)$ das gleiche Vorzeichen wie $f(x_2)$?

- ersetze entsprechende Intervallgrenze durch x_2
- Intervall wird halb so groß
- wiederhole bis gewünschte Genauigkeit erreicht

Mathematische Funktionen

Mathematische Funktionen

`Math.PI` bzw. `Math.E` definieren den Wert von π bzw. e

„Funktion“	Bezeichnung	Ergebnis
`Math.abs(`*x*`)`	Absolutwert	|*x*|
`Math.sqrt(`*x*`)`	Quadratwurzel	$\sqrt{x}$
`Math.pow(`*x*`,`*y*`)`	Potenz	x^y
`Math.exp(`*x*`)`	Exponentialfunktion	e^x
`Math.log(`*x*`)`	natürlicher Logarithmus	ln*x*
`Math.sin(`*x*`)`	Sinus	sin*x*
`Math.atan(`*x*`)`	inverser Tangens	$\tan^{-1}x$

etc.

Mathematische Funktionen

Beispiel:

```
// Datei: Beispiel4.java

public class Beispiel4 {
      public static void main(String[] args) {
      System.out.println("cos(Pi/3) = " + Math.cos(Math.PI/3));
      System.out.println("sin(Pi/3) = " + Math.sin(Math.PI/3));
      System.out.println("sqrt(3)/2 = " + Math.sqrt(3)/2);
      System.out.println("tanh(10) = " + Math.tanh(10));
   }
}
```

Ergebnis:

```
cos(Pi/3) = 0.5000000000000001
sin(Pi/3) = 0.8660254037844386
sqrt(3)/2 = 0.8660254037844386
tanh(10) = 0.9999999958776927
```