Eine Anweisung

wird mit dem Wertzuweisungsoperator $=$ geschrieben.

Eine Anweisung

wird mit dem Wertzuweisungsoperator $=$ geschrieben.

Daher ist

$$y = x + 5.6;$$

keine Gleichung, sondern die Anweisung den Wert aus der Speicherzelle von x zu holen, zu 5.6 zu addieren und in der Speicherzelle von y abzuspeichern.

Die Arithmetischen Operatoren

$+$ $-$ $*$ $/$ $\%$ wirken, wie aus der Mathematik bekannt.

Der Divisionsoperator $/$ liefert im Falle zweier ganzzahliger Operanden den ganzzahligen Anteil.

Den Divisionsrest liefert der Modulo-Operator $\%$.

$+$ und $-$ werden auch wie üblich als Vorzeichen benutzt.

Die Arithmetischen Operatoren

$+$ $-$ $*$ $/$ $\%$ wirken, wie aus der Mathematik bekannt.

Der Divisionsoperator $/$ liefert im Falle zweier ganzzahliger Operanden den ganzzahligen Anteil.

Den Divisionsrest liefert der Modulo-Operator $\%$.

$+$ und $-$ werden auch wie üblich als Vorzeichen benutzt.

Ergänzen Sie Ihr Programm so, dass für zwei ganzzahligen Variablen vom Typ (int) mit den Anfangswerten 5 und 7 deren Summe, Differenz, Produkt und Quotient berechnet und ausgegeben werden.

Vergleichs-Operatoren

Um zu entscheiden, ob Vergleiche den boolschen Wert false oder true liefern, werden Vergleichsoperatoren benötigt.

Vergleichs-Operatoren

Um zu entscheiden, ob Vergleiche den boolschen Wert false oder true liefern, werden Vergleichsoperatoren benötigt.
Seien x und y Objekte des gleichen Grunddatentypes:

Operator	Bedeutung
$x==y$	true, falls x und y gleich sind.
$x!=y$	true, falls x und y ungleich sind.
$x<y$	true, falls x kleiner y ist.
$x<=y$	true, falls x kleiner oder gleich y ist.
$x>y$	true, falls x größer y ist.
$x>=y$	true, falls x größer oder gleich y ist.

Vergleichs-Operatoren

Mit den Vergleichsoperatoren können wir einfache Vergleiche abtesten. Wie aber testen wir z.B., ob

$$0 \leq x \leq 7$$

gilt?

Vergleichs-Operatoren

Mit den Vergleichsoperatoren können wir einfache Vergleiche abtesten. Wie aber testen wir z.B., ob

$$0 \leq x \leq 7$$

gilt?
Hier müssen wir testen, ob

$$0 \leq x \text{ und gleichzeitig } x \leq 7$$

ist. Dazu brauchen wir die Verknüpfung logischer Ausdrücke.

Logische Operatoren

Logische Operatoren ! | & ^ && || erlauben es, Verknüpfungen zwischen logischen Werten herzustellen und somit komplexe logische Ausdrücke zu erstellen.

Logische Operatoren

Logische Operatoren ! | & ^ && || erlauben es, Verknüpfungen zwischen logischen Werten herzustellen und somit komplexe logische Ausdrücke zu erstellen.

Dabei gilt, falls $bool_1$ und $bool_2$ logische Ausdrücke sind

Operator	Bedeutung
! $bool_1$	Negation von $bool_1$
$bool_1$ | $bool_2$	Logisches Oder, true wenn $bool_1$ oder $bool_2$ true sind.
$bool_1$ & $bool_2$	Logisches Und, true wenn $bool_1$ und $bool_2$ true sind.

Logische Operatoren

Logische Operatoren ! | & ^ && || erlauben es, Verknüpfungen zwischen logischen Werten herzustellen und somit komplexe logische Ausdrücke zu erstellen.

Die anderen drei Operatoren werden seltener benötigt - bitte im Selbststudium erarbeiten!

Logische Operatoren

Logische Operatoren ! | & ^ && || erlauben es, Verknüpfungen zwischen logischen Werten herzustellen und somit komplexe logische Ausdrücke zu erstellen.

Der logische Ausdruck unseres Beispiels $0 \leq x \leq 7$ würde nun lauten:
`0 <= x & x <= 7`.

Logische Operatoren

Der logische Ausdruck unseres Beispiels $0 \leq x \leq 7$ würde nun lauten:

`0 <= x & x <= 7`.

Es ist nicht verkehrt Klammern zu setzen:

`(0 <= x) & (x <= 7)`

Logische Operatoren

Formulieren Sie einen logischen Ausdruck für die Tatsache, dass höchstens doppelt so viele Studenten im Raum sind wie Computer, aber mindestens ein Student da ist.

Logische Operatoren

Formulieren Sie einen logischen Ausdruck für die Tatsache, dass höchstens doppelt so viele Studenten im Raum sind wie Computer, aber mindestens ein Student da ist.

student $<=$ 2$*$computer & student $>$ 0

Blöcke und Kontrollstrukturen

- Ein Block enthält eine Folge von Anweisungen
- Die for- oder Zählschleife
- Die if-else-Anweisung

Blöcke und Kontrollstrukturen

- Ein Block enthält eine Folge von Anweisungen,

  die in { ... } eingeschlossen werden. Die Anweisungen werden nacheinander ausgeführt.

```
{
        Anweisung_1;
        Anweisung_2;
        :
        Anweisung_n;
}
```

- Die for- oder Zählschleife
- Die if-else-Anweisung

Blöcke und Kontrollstrukturen

- Ein Block enthält eine Folge von Anweisungen

  Ein Block wirkt nach aussen wie eine Anweisung.
- Die for- oder Zählschleife
- Die if-else-Anweisung

Blöcke und Kontrollstrukturen

- Ein Block enthält eine Folge von Anweisungen

  Ein Block wirkt nach aussen wie eine Anweisung.

  Blöcke können geschachtelt werden.
- Die for- oder Zählschleife
- Die if-else-Anweisung

Blöcke und Kontrollstrukturen

- Ein Block enthält eine Folge von Anweisungen

  Ein Block wirkt nach aussen wie eine Anweisung.

  Blöcke können geschachtelt werden.

  In einem Block definierte Objekte sind nur in diesem Block bekannt!
- Die for- oder Zählschleife
- Die if-else-Anweisung

Blöcke und Kontrollstrukturen

- Ein Block enthält eine Folge von Anweisungen
- Die for- oder Zählschleife
- Die if-else-Anweisung

Blöcke und Kontrollstrukturen

- Ein Block enthält eine Folge von Anweisungen
- Die for- oder Zählschleife hat die Struktur

  for(Initializierungsanw.; boolescher Ausdruck; Inkrementanw.)
  Anweisung(sblock);

- Die if-else-Anweisung

Blöcke und Kontrollstrukturen

- Ein Block enthält eine Folge von Anweisungen
- Die for- oder Zählschleife

  for(Initializierungsanw.; boolescher Ausdruck; Inkrementanw.)
  Anweisung(sblock);

  Die Abarbeitung ist:

  1. Ausführen der Initializierungsanweisung
  2. falls der boolescher Ausdruck true ist, werden die Anweisung und dann die Inkrementanweisung ausgeführt,
     sonst die for-Schleife beendet
  3. weiter bei 2.
- Die if-else-Anweisung

Blöcke und Kontrollstrukturen

- Ein Block enthält eine Folge von Anweisungen
- Die for- oder Zählschleife

  Die Abarbeitung ist:

  1. Ausführen der Initializierungsanweisung
  2. falls der boolescher Ausdruck true ist, werden die Anweisung und dann die Inkrementanweisung ausgeführt, sonst die for-Schleife beendet
  3. weiter bei 2.

  Beispiel: `int j=0;`
  `for( int i=1; i < 11; i=i+1) j=j+i;`
- Die if-else-Anweisung

Blöcke und Kontrollstrukturen

- Ein Block enthält eine Folge von Anweisungen
- Die for- oder Zählschleife
- Die if-else-Anweisung

Blöcke und Kontrollstrukturen

- Ein Block enthält eine Folge von Anweisungen
- Die for- oder Zählschleife
- Die if-else-Anweisung hat die Struktur

```
if (boolescher Ausdruck)
                Anweisung_1;
else
                Anweisung_2;
```

Blöcke und Kontrollstrukturen

- Ein Block enthält eine Folge von Anweisungen
- Die for- oder Zählschleife
- Die if-else-Anweisung

```
if (boolescher Ausdruck)
                Anweisung_1;
else
                Anweisung_2;
```

Die Abarbeitung wird durch den booleschen Ausdruck gesteuert.
Ist der booleschen Ausdruck = true wird
Anweisung_1 ausgeführt, sonst
Anweisung_2.

Blöcke und Kontrollstrukturen

- Ein Block enthält eine Folge von Anweisungen
- Die for- oder Zählschleife
- Die if-else-Anweisung

  Die Abarbeitung wird durch den booleschen Ausdruck gesteuert.
  Ist der booleschen Ausdruck = true wird
  Anweisung_1 ausgeführt, sonst
  Anweisung_2.
  Beispiel:

```
if (student > 0)
    System.out.println("\u00dcbung findet statt!");
else
    System.out.println("Keine \u00dcbung!");
```

Ü

Blöcke und Kontrollstrukturen

- Ein Block enthält eine Folge von Anweisungen
- Die for- oder Zählschleife
- Die if-else-Anweisung

```
if (boolescher Ausdruck)
                Anweisung_1;
else
                Anweisung_2;
```

Merke: Der else-Zweig kann fehlen!

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
- Import von Bibliotheken
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
  Sehr viele nützliche Programme liegen schon vor und wir wollen sie benutzen. Dazu gibt es Bibliotheken.
  WRI-Homepage → vorhandene Bibliotheken → JAVA_2 → links oben java.lang → links unten Math
- Import von Bibliotheken
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
  Alles in java.lang kann direkt benutzt werden. Nur wie?
- Import von Bibliotheken
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
  Alles in java.lang kann direkt benutzt werden. Nur wie?
  Die ausgewählte Klasse heisst Math und wir wollen eine statische Methode (Unterprogramm) benutzen.
- Import von Bibliotheken
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
  Die ausgewählte Klasse heisst Math und wir wollen eine statische Methode (Unterprogramm) benutzen.
  → Slider am rechten Rand bis zur Methode sqrt im Method Summary herunterziehen.
- Import von Bibliotheken
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
  An der ersten Spalte erkennt man, dass alle diese Methoden static sind. (Was das genau ist, folgt später.)
- Import von Bibliotheken
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
  An der ersten Spalte erkennt man, dass alle diese Methoden static sind. (Was das genau ist, folgt später.)

  Lesen Sie die Beschreibung von sqrt.
- Import von Bibliotheken
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
  An der ersten Spalte erkennt man, dass alle diese Methoden static sind. (Was das genau ist, folgt später.)

  Lesen Sie die Beschreibung von sqrt.

  Die statische Methode methode mit dem Ergebnistyp typ in der Klasse kl werden unter Berücksichtigung ihrer Parameterliste folgendermaßen aufgerufen:

  typ erg=kl.methode(<Var. entsp. der Parameterliste>);
- Import von Bibliotheken
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
  An der ersten Spalte erkennt man, dass alle diese Methoden static sind. (Was das genau ist, folgt später.)

  Lesen Sie die Beschreibung von sqrt.

  Die statische Methode methode mit dem Ergebnistyp typ in der Klasse kl werden unter Berücksichtigung ihrer Parameterliste folgendermaßen aufgerufen:

  typ erg=kl.methode(<Var. entsp. der Parameterliste>);

  Beispiel: double rad=2.0;
  double wurzel=Math.sqrt(rad);
- Import von Bibliotheken
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
  Analog sind auch Konstanten einer Klasse (→ Field Summary) zu benutzen:

  Beispiel: `double pi=Math.PI;`
- Import von Bibliotheken
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
- Import von Bibliotheken
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
- Import von Bibliotheken
  → vorhandene Bibliotheken → HUMath →
  HUMath.Numerik→ ReadDialog
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
- Import von Bibliotheken
  Über der Klassenbezeichnung ReadDialog sehen wir den package-Namen HUMath.Numerik.
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
- Import von Bibliotheken
  Über der Klassenbezeichnung ReadDialog sehen wir den package-Namen HUMath.Numerik.

  Wenn wir Methoden aus dieser Klasse in unserem Programm benutzen wollen, müssen wir sie importieren.
  Wir schreiben am Anfang unseres java-Programmes <name>.java die Zeile(n):

  import HUMath.Numerik.ReadDialog; bzw.

  import HUMath.Numerik.*;
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
- Import von Bibliotheken
  Dann können wir die statischen Methoden aus der Klasse ReadDialog zum Einlesen von Daten benutzen:

  Beispiel: `double rad = ReadDialog.getDouble("Radius = ");`
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
- Import von Bibliotheken
  Bemerkung:
  Beenden Sie Ihr Programm (bei der Nutzung graphischer Elemente) mit System.exit(0);
  oder per Hand mit ^C.
-

Benutzen *statischer* Methoden aus Bibliotheken

- Benutzen statischer Methoden
- Import von Bibliotheken
- Jetzt können die ersten Java-Programme entstehen!

Benutzen von (externen) Bibliotheken

- Der import von Bibliotheken wurde schon besprochen.

  Damit zusätzliche Bibliotheken verwendet werden können, müssen sie für Java erreichbar sein.

  Man sagt: Die Bibliotheken müssen im Pfad stehen.

  Überprüfung: echo $CLASSPATH

-

Benutzen von (externen) Bibliotheken

- Der import von Bibliotheken wurde schon besprochen.

  Damit zusätzliche Bibliotheken verwendet werden können, müssen sie für Java erreichbar sein.

  Man sagt: Die Bibliotheken müssen im Pfad stehen.

  Überprüfung: echo $CLASSPATH
- Merke: Was man lesen kann, kann man auch kopieren!

Typumwandlungen und Kurzformen

- Typumwandlungen
  Manchmal benötigt man eine Variable in einem anderen Typ.

  Schreibt man: `double x = 47.11;`
  `int i = x;`
- Kompakte Schreibweisen sind hilfreich.
- "Überschreiben" von Variablen
- Inkrement (++) und Dekrement (--) Operator

Typumwandlungen und Kurzformen

- Typumwandlungen
  Manchmal benötigt man eine Variable in einem anderen Typ.

  Schreibt man: `double x = 47.11;`
  `int i = x;`

  bekommt man folgende Fehlermeldung:

  ```
  format.java:12: possible loss of precision
  found : double
  required: int
  int i = x;
          ^
  1 error
  ```
- Kompakte Schreibweisen sind hilfreich.
- "Überschreiben" von Variablen
- Inkrement (++) und Dekrement (--) Operator

Typumwandlungen und Kurzformen

- Typumwandlungen
  Daher ist eine Typumwandlung nötig. Man wendet den cast-Operator an:

  (neuerTyp) Objekt

  Beispiel: `int i = (int) x;`
- Kompakte Schreibweisen sind hilfreich.
- "Überschreiben" von Variablen
- Inkrement (++) und Dekrement (--) Operator

Typumwandlungen und Kurzformen

- Typumwandlungen
- Kompakte Schreibweisen sind hilfreich.
  Mehrfacher Zuweisungsoperator: a = b = c; bedeutet
- "Überschreiben" von Variablen
- Inkrement (++) und Dekrement (--) Operator

Typumwandlungen und Kurzformen

- Typumwandlungen
- Kompakte Schreibweisen sind hilfreich.
  Mehrfacher Zuweisungsoperator: `a = b = c;` bedeutet

  ```
  b = c;
  a = b;
  ```

- "Überschreiben" von Variablen
- Inkrement (++) und Dekrement (--) Operator

Typumwandlungen und Kurzformen

- Typumwandlungen
- Kompakte Schreibweisen sind hilfreich.
  Mehrfacher Zuweisungsoperator: `a = (b = c);` bedeutet

  `b = c;`
  `a = b;` (rechts assoziativ)
- "Überschreiben" von Variablen
- Inkrement (++) und Dekrement (--) Operator

Typumwandlungen und Kurzformen

- Typumwandlungen
- Kompakte Schreibweisen sind hilfreich.
- "Überschreiben" von Variablen
- Inkrement (++) und Dekrement (--) Operator

Typumwandlungen und Kurzformen

- Typumwandlungen
- Kompakte Schreibweisen sind hilfreich.
- "Überschreiben" von Variablen
  Oft haben Anweisungen die Form $x = x$ op y;
- Inkrement (++) und Dekrement (--) Operator

Typumwandlungen und Kurzformen

- Typumwandlungen
- Kompakte Schreibweisen sind hilfreich.
- "Überschreiben" von Variablen
  Oft haben Anweisungen die Form x = x op y;

  Beispiel: x = x / y;

  Kurzform: x /= y; oder allgemein
- Inkrement (++) und Dekrement (--) Operator

Typumwandlungen und Kurzformen

- Typumwandlungen
- Kompakte Schreibweisen sind hilfreich.
- "Überschreiben" von Variablen
  Typ objekt op = Ausdruck;

  was bedeutet: Typ objekt = (Typ)(objekt op Ausdruck);
- Inkrement (++) und Dekrement (--) Operator

Typumwandlungen und Kurzformen

- Typumwandlungen
- Kompakte Schreibweisen sind hilfreich.
- "Überschreiben" von Variablen
- Inkrement (++) und Dekrement (--) Operator

  erhöht bzw. erniedrigt die Variable um 1.

Typumwandlungen und Kurzformen

- Typumwandlungen
- Kompakte Schreibweisen sind hilfreich.
- "Überschreiben" von Variablen
- Inkrement (++) und Dekrement (--) Operator

  erhöht bzw. erniedrigt die Variable um 1.

  In der Form x ++ (auch x++) nachdem er benutzt wird

  und in der Form ++ x bevor er benutzt wird.

Typumwandlungen und Kurzformen

- Typumwandlungen
- Kompakte Schreibweisen sind hilfreich.
- "Überschreiben" von Variablen
- Inkrement (++) und Dekrement (--) Operator

  erhöht bzw. erniedrigt die Variable um 1.

  In der Form x ++ (auch x++) nachdem er benutzt wird

  und in der Form ++ x bevor er benutzt wird.

  Beispiel: `int x = 0; int y = x++ +2;` Nach der Ausführung ist x = 1 und y = 2.

Typumwandlungen und Kurzformen

- Typumwandlungen
- Kompakte Schreibweisen sind hilfreich.
- "Überschreiben" von Variablen
- Inkrement (++) und Dekrement (--) Operator

  erhöht bzw. erniedrigt die Variable um 1.

  In der Form x ++ (auch x++) nachdem er benutzt wird

  und in der Form ++ x bevor er benutzt wird.

  Beispiel: `int x = 0; int y = x++ +2;` Nach der Ausführung ist x = 1 und y = 2.

  `int x = 0; int y = ++x +2;` Nach der Ausführung ist x = 1 und y = 3.

Typumwandlungen und Kurzformen

- Typumwandlungen
- Kompakte Schreibweisen sind hilfreich.
- "Überschreiben" von Variablen
- Inkrement (++) und Dekrement (--) Operator

  erhöht bzw. erniedrigt die Variable um 1.

  Oft wird nur der Befehl `x++;` benutzt.

Die while-Anweisung

Häufig sollen Anweisungen abhängig von einer Bedingung abgearbeitet werden.

```
while ( boolescher Ausdruck )
        Anweisung;
```

Die while-Anweisung

Häufig sollen Anweisungen abhängig von einer Bedingung abgearbeitet werden.

while (boolescher Ausdruck)
Anweisung;

Die Anweisung wird ausgeführt,
solange der boolescher Ausdruck = true ist.

Die while-Anweisung

Häufig sollen Anweisungen abhängig von einer Bedingung abgearbeitet werden.

```
while ( boolescher Ausdruck )
        Anweisung;
```

Zweite Variante (nicht abweisende while-Schleife):

```
do
        Anweisung;
while ( boolescher Ausdruck );
```

Die while-Anweisung

Häufig sollen Anweisungen abhängig von einer Bedingung abgearbeitet werden.

```
while ( boolescher Ausdruck )
        Anweisung;

do
        Anweisung;
while ( boolescher Ausdruck );
```

Die Anweisung wird mindestens einmal ausgeführt und, solange der boolescher Ausdruck = true ist, fortgefahren.

Die switch-Anweisung

dient der Ausführung von Anweisungen, wenn mehrere Bedingungen berücksichtigt werden sollen.

Die switch-Anweisung

```
switch ( Ausdruck ) {
        case Konstante_1 :
                Anweisung_1;
                break;
        case Konstante_2 :
                Anweisung_2;
                break;
        :
        default:
                DefaultAnweisung;
}
```

Die switch-Anweisung

```
switch ( Ausdruck ) {
        case Konstante_1 :
                Anweisung_1;
                break;
        case Konstante_2 :
                Anweisung_2;
                break;
        :
        default:
                DefaultAnweisung;
}
```

Typ: byte, char, short, int, long

Die switch-Anweisung

```
switch ( Ausdruck ) {
        case Konstante_1 :
                Anweisung_1;
                break;
        case Konstante_2 :
                Anweisung_2;
                break;
        :
        default:
                DefaultAnweisung;
}
```

Typ: byte, char, short, int, long
vom Typ des Ausdrucks

Die switch-Anweisung

```
switch ( Ausdruck ) {
        case Konstante_1 :
                Anweisung_1;
                break;
        case Konstante_2 :
                Anweisung_2;
                break;
        :
        default:
                DefaultAnweisung;
}
```

Typ: byte, char, short, int, long
vom Typ des Ausdrucks

falls break fehlt, fall through

Die switch-Anweisung

```
switch ( Ausdruck ) {
        case Konstante_1 :
                Anweisung_1;
                break;
        case Konstante_2 :
                Anweisung_2;
                break;
        :
        default:
                DefaultAnweisung;
}
```

Typ: byte, char, short, int, long
vom Typ des Ausdrucks

falls break fehlt, fall through

falls Ausdruck $\neq$ Konstante_i, $\forall i$

Beispiel

```
int i, erg;
:
switch (i){
        case 1:
            erg=1;
            break;
        case 6:
            erg=2;
            break;
        default:
            erg=0;
}
```