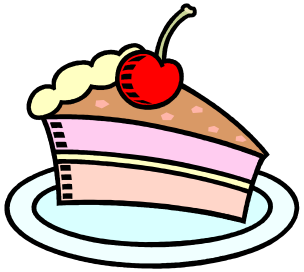
Algorithmische Kernsprache

Zuweisung, einfache und bedingte Anweisung, Blöcke, Schleifen, return, debugging.

Ausdrücke – Anweisungen

- Ausdrücke bezeichnen einen Wert
  - Kontext stellt Werte von Variablen
  - Werte werden mit Operationen verknüpft
  - der Kontext wird dabei nicht verändert



Ausdrücke (Beispiele):	*Wert*
`5-7+44`	`- 42 – in jedem Kontext`
`x*x + y*y + 1`	`- eine Zahl ≥ 0 – je nach Kontext`
`(2<3)? (17+4) : y+1`	`- 21`
`meinKonto.getKontoStand()`	`- mein Kontostand`
`new Triangle()`	`- ein Objekt der Klasse Triangle`

©H. Peter Gumm, Philipps-Universität Marburg

Ausdrücke – Anweisungen

- Anweisungen verändern einen Kontext
  - Anweisungen liefern *keinen* Wert
  - Anweisungen haben einen Effekt

Anweisungen (Beispiele):

```
System.out.println("Hallo Welt");

x=x+1;

if (x<0) x = -x;

circle1.move();

meinKonto.abheben(meinKonto.getKontoStand());
```

Effekt

- schreibt „Hallo Welt“ auf die Konsole
- erhöht x um 1
- setzt x auf Absolutwert
- bewegt circle1
- räumt mein Konto leer

©H. Peter Gumm, Philipps-Universität Marburg

Anweisung oder Ausdruck?

- Welche der folgenden Bestandteile eines Java-Programmes sind Anweisungen, welche sind Ausdrücke ?
  - `System.out.println(x+y+1);`
  - `meinKonto.getKontostand() + deinKonto.getKontostand()`
  - `new Konto(“Gumm“,100)`
  - `Konto meins = new Konto(“Gumm“,100);`
  - `“Summe = “ + resultat + “;“`
  - `Math.max(Math.sqrt(x),x/2)`
  - `new Konto(“Meier“,100)`
  - `{ int x=10; x=x+1; }`
  - `( x+x+1 )`

… zugegeben, das Semikolon ist verräterisch; später müssen Sie es selber setzen.

©H. Peter Gumm, Philipps-Universität Marburg

Zuweisung – die einfachste Anweisung

- Syntax:

```
v = E;
```

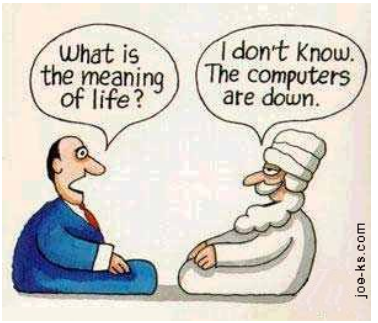
  - v eine vorher deklarierte Variable
  - E ein zum Typ von v kompatibler Ausdruck

- Beispiele

```
x = 5 ;        // Wert wird gespeichert
x = 3*y+1 ;    // Wertberechnung und Speicherung
y = x+y ;      // Variable darf rechts auch vorkommen
z = z+1 ;      // Inkrement
```


©H. Peter Gumm, Philipps-Universität Marburg

Zuweisung - Semantik



Semantik:

```
v = E;
```

1. berechne den Wert von E im aktuellen Kontext
2. speichere den gefundenen Wert in v

```
betrag = betrag*(100+zinsSatz)/100 ;
```

int-Variable. Hier wird der Wert gespeichert

Zuweisungs-operator

Ausdruck, liefert Wert vom Typ int

Semikolon

©H. Peter Gumm, Philipps-Universität Marburg

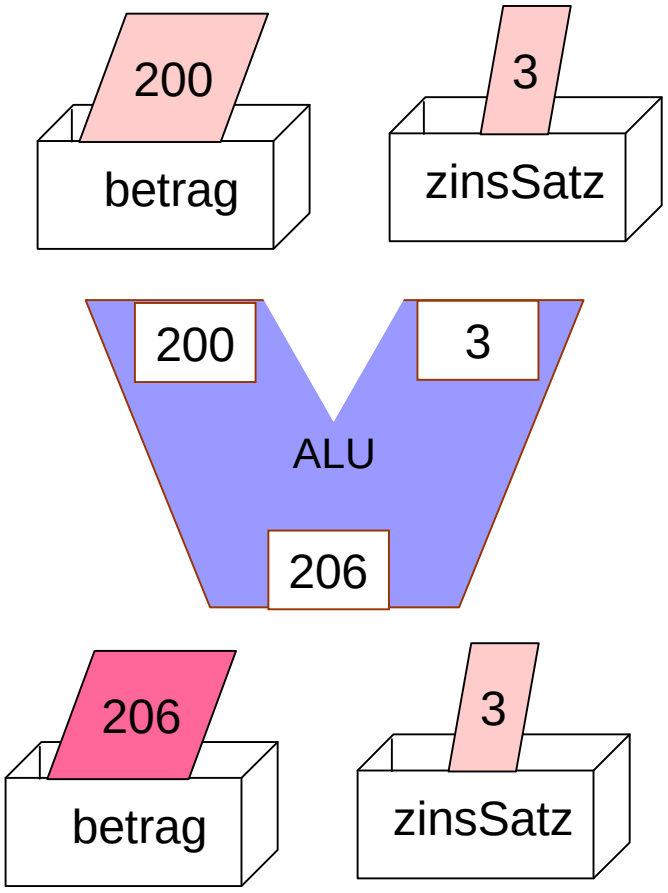
Zuweisung – die Aktion der CPU

```
betrag = betrag*(100+zinsSatz)/100 ;
```

- 1. Phase
  Inhalt von **betrag** und **zinsSatz** *lesen*
- 2. Phase
  Mit den gefundenen Werten
  im aktuellen Kontext
  den Wert des Ausdrucks

  betrag*(100+zinsSatz)/100

  berechnen
- 3. Phase
  Den berechneten Wert in der Variablen
  betrag
  speichern



©H. Peter Gumm, Philipps-Universität Marburg

Syntaktische Varianten der Zuweisung

- Syntaktische Varianten
  - In/Decrement-Anweisungen:
    - `v += E;  v -= E; v *= E; v /= E; v %= E;`

      stehen der Reihe nach für

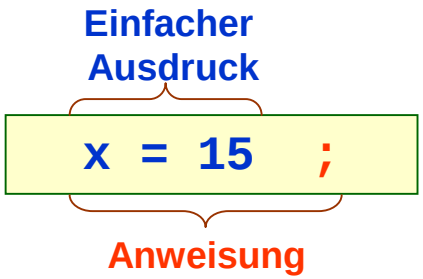
      `v = v+E; v=v-E; v = v*E; v=v/E; v= v%E;`
  - Auto-In/Decrement :
    - `v++ ;` bzw. `v-- ;`

      stehen für `v = v+1;` bzw. `v=v-1;`

©H. Peter Gumm, Philipps-Universität Marburg

Was soll das Semikolon ; ?

- Java vermischt Anweisungen und Ausdrücke
  - ***Einfache Ausdrücke*** haben einen Effekt, z.B.:
    - x = 15 (kein Semikolon !!!)
    - x++
    - return 3*x+1
    - überweisen(23,deinKonto)



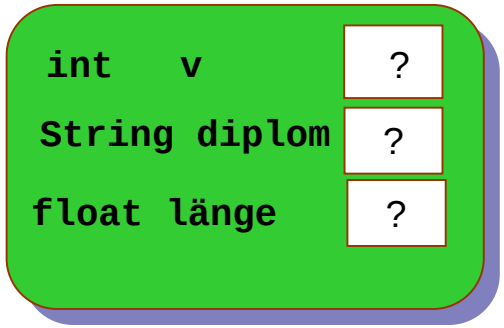
  - Sie dürfen überall stehen, wo Ausdrücke erlaubt sind:
    - ```
      if (x=15 != y)
            System.out.println(x=15) ;
      ```
    - ```
      while((x++ - y--) != 0)
            System.out.printl(x=y=z=x++)
      ```
    - Derartiges wollen wir in den Übungsaufgaben **nicht** sehen.
      - Wird als Fehler gewertet
  - Ein Semikolon macht aus einem *einfachen Ausdruck* eine *Anweisung*

©H. Peter Gumm, Philipps-Universität Marburg

Java: Deklaration ist Anweisung

- Eine Deklaration erweitert einen Kontext
  - Variable wird deklariert, indem man ihr den Typ vorausstellt.
  - Deklaration endet mit einem Semikolon.

```
int v ;
String diplom ;
float länge ;
```



  - Man darf mehrere Variablen des gleichen Typs gemeinsam deklarieren

```
int v, kontoStand, xPos ;
String notizen, diplomArbeit ;
float länge, breite, höhe ;
```

- Im gleichen Kontext darf gleichnamige Variable noch nicht erklärt sein

```
{
  int zinsSatz;
   float zinssatz ;    // legal, aber fies –
   double zinsSatz ;   // illegal, schon erklärt.
}
```

- Deklaration initialisiert Variable nicht

©H. Peter Gumm, Philipps-Universität Marburg

Initialisierung

- In den meisten Sprachen gilt:
  - Nach der Deklaration hat eine Variable einen zufälligen Inhalt
    - Was gerade an der Stelle im Speicher herumlag …
- In Java wird zwar jedes Feld einer Klassendefinition automatisch mit dem „default“-Wert des Typs initialisiert.
  - `byte, short, int, long:  0`
  - `boolean :  false`
  - `float, double :  0.0`
  - `char :  \u0000`
  - alle Objekt-Typen: `null`

`int n`	5
`boolean fertig`	false
`String Name`	“Ey“

- Dennoch wird empfohlen, Felder zu initialisieren:
  - `int anzahl = 0;`
  - `boolean voll = false;`
  - `int summe = 0;`
  - `String Name = “Otto“;`
- Lokale Variablen in Methodenrümpfen müssen explizit initialisiert werden, bevor sie gelesen werden.

```
void test()
  {
    int y;
    y = x + y;  // Fehler !!
  }
```

variable y might not have been initialized

©H. Peter Gumm, Philipps-Universität Marburg

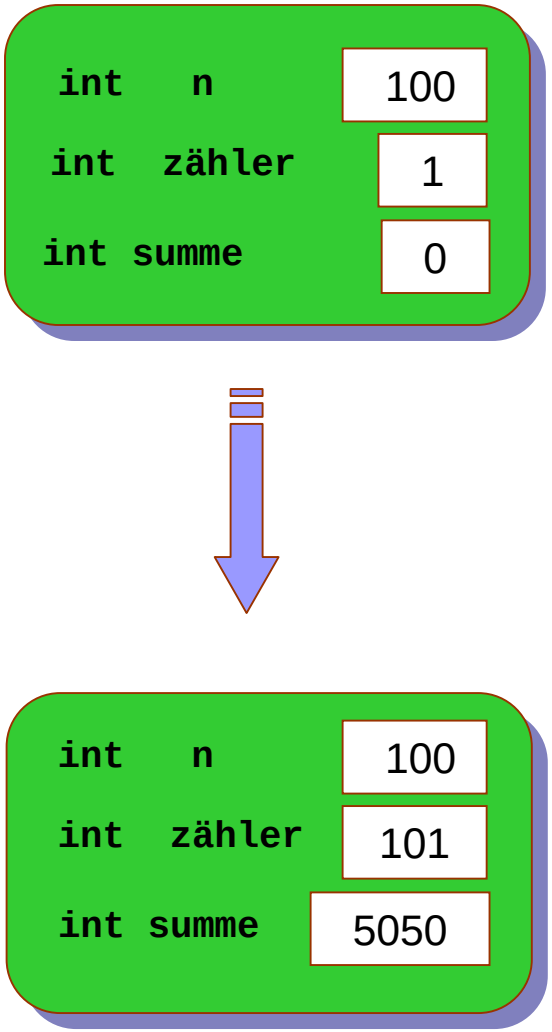
Mehrere Anweisungen

- Anweisungen dienen dazu, einen Kontext/Zustand gezielt zu verändern
  - eine Zuweisung reicht meist nicht aus
  - mehrere Anweisungen werden zu einer komplexen Anweisung kombiniert
- Beispiel:
  Berechne die Summe aller Zahlen von 1 bis n=100:
  - Beginne mit dem Kontext

    ```
    { int n = 100, zähler=0, summe=0; }
    ```

  - Verändere den Kontext, indem in jedem Schritt
    - **zähler** zu **summe** addiert wird
    - **zähler** erhöht wird
  - ... bis **zähler** größer als **n** ist.

  Das gesuchte Ergebnis ist der Wert von **summe** im entstandenen Kontext.



©H. Peter Gumm, Philipps-Universität Marburg

Kontrollstrukturen

- Kombinationen von Anweisungen
  - *Hintereinanderausführung*
    - { A_1 A_2 A_3 } erst A_1 dann A_2 dann A_3
  - *Alternativanweisung*
    - falls B dann A_1 sonst A_2
  - *Schleifen*
    - für alle x von 1 bis 100 : A_1
    - solange x*x<100 : erhöhe x um 1
    - schreibe “Halt mich“ bis 2 < 1
- Im Prinzip reichen diese drei Kontrollstrukturen
  - in der Praxis ist es aber nützlich, weitere zu haben
    - Fallunterscheidung,
    - bedingte Anweisung
    - return, break, continue, …

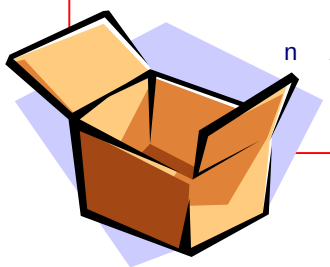
©H. Peter Gumm, Philipps-Universität Marburg

Hintereinanderausführung - Blöcke

- Blöcke sind „*Programm-Schachteln*“
  - *Block* : Folge von Anweisungen

$$\{\ Anw_1\ \ Anw_2\ \ Anw_3\ \ \ldots\ \}$$

  - *Block* ist Anweisung
    - daher Schachtelung möglich
  - *Block* definiert *neuen Kontext*
    - Variablen können (erneut) definiert werden
    - … beim Verlassen des Blockes endet ihre Lebensdauer, sie sind nicht mehr sichtbar.
    - Außen definierte Variablen dürfen innen *nicht* neu definiert werden ... in vielen anderen Sprachen wäre das möglich



```
{ // Ein Block
  int v=5;
  int summe=0;
  summe += v*v;
  v++;
  summe += v*v;
}
```

```
{ // äusserer Block
int v=5;
int summe=0;
summe += v*v;
{ // innerer Block
  int temp = summe+v ;
  int  v = 3; //illegal
}
summe += temp;
}
```

v is already defined in altTest()

?

gespeichert

Das ist in Java (leider) nicht legal.

Fehler, da **`temp`** hier nicht mehr sichtbar.

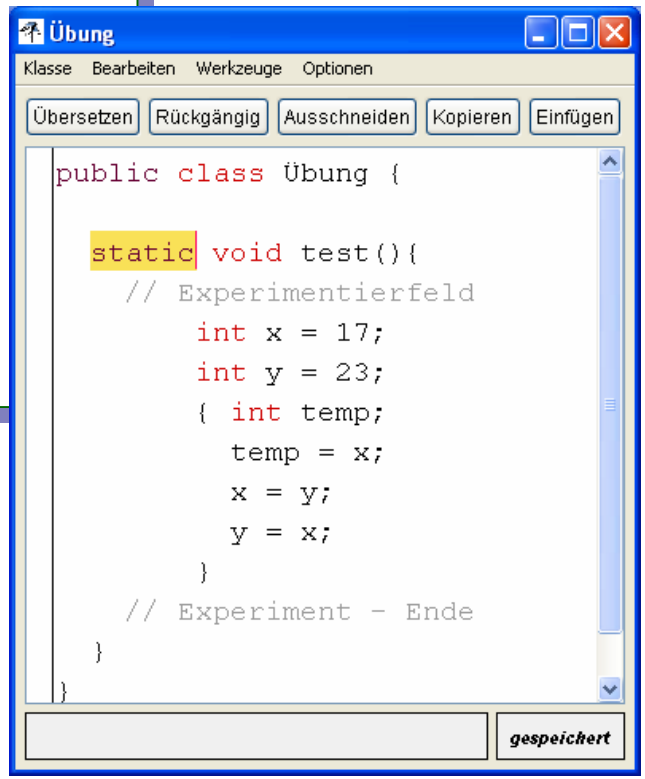
©H. Peter Gumm, Philipps-Universität Marburg

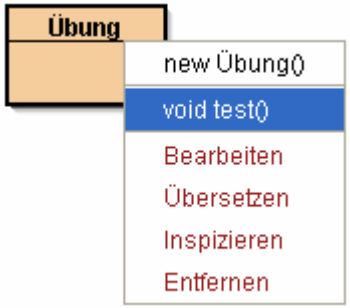
Experimente mit BlueJ

- Definieren Sie eine Klasse **Übung** mit einer Klassenmethode test:

```
static void test( ){
        …
}
```

- Im Rumpf der Methode können Sie experimentieren.
- Compilieren Sie das Programm
- Rufen Sie **`test()`** aus dem Kontextmenü der Klasse **Übung** auf.

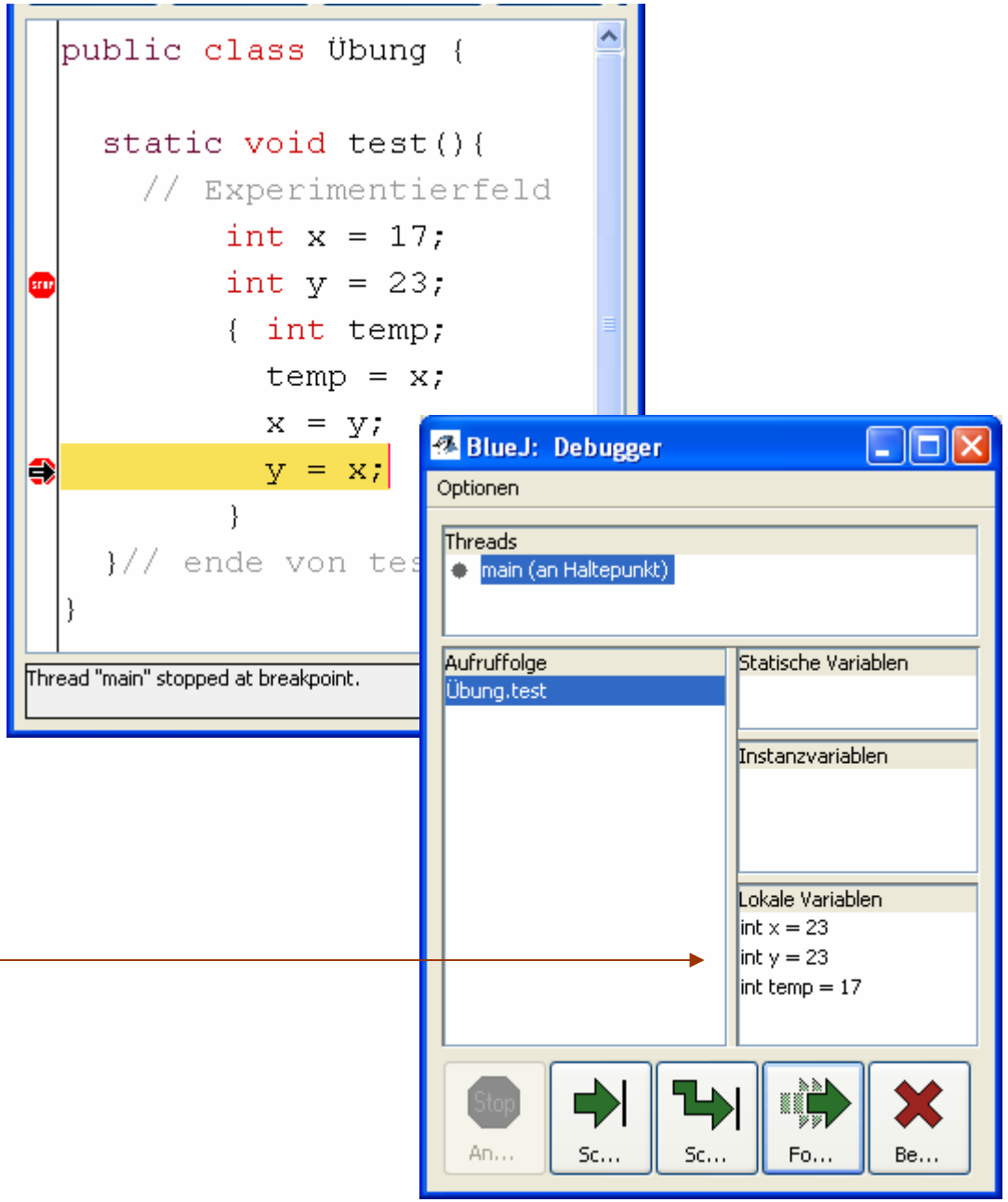




©H. Peter Gumm, Philipps-Universität Marburg

Debugging

- Durch Klicken in die linke Spalte des Editors setzen Sie – nach dem Compilieren – *Breakpoints*.
- Rufen Sie **`test()`** aus dem Kontextmenü der Klasse **Übung**
- Beim nächsten Aufruf von **`test()`** erscheint ein *Debugger*.
- Damit können Sie das Programm schrittweise durchgehen (*Step* oder *Step Into*)
- Bei jedem *Breakpoint* hält das Programm
- Im *Debuggerfenster* sehen Sie den aktuellen Inhalt aller Variablen



©H. Peter Gumm, Philipps-Universität Marburg

Alternativanweisung

- Je nach dem ob eine Bedingung wahr oder falsch ist, wird eine oder eine andere Anweisung ausgeführt

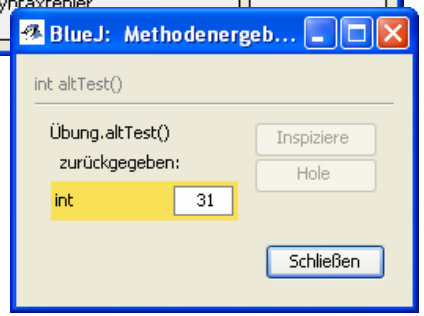
- Syntax:

```
if ( Bedingung ) Anweisung1
else Anweisung2
```

  - *Bedingung* muss boolescher Ausdruck sein
  - *Anweisung*$_1$ und *Anweisung*$_2$ sind beliebige Anweisungen, also z.B.:
    - Zuweisungen
    - Blöcke
    - bedingte oder Alternativ-Anweisungen
    - …

```
static int altTest(){
   // Alternativanweisung
    int v = 5;
    int summe = 0;
    if( v > 0 ) {
        summe += v*v;
        v++;
      }
    else summe = 42;
    return summe+v;
}// ende von altTest
```

Klasse übersetzt - keine Syntaxfehler



©H. Peter Gumm, Philipps-Universität Marburg

Bedingte Anweisung

- Nur falls eine bestimmte *Bedingung* **`true`** ist, wird die Anweisung ausgeführt
- Syntax:

```
if ( Bedingung ) Anweisung
```

  - *Bedingung* ein boolescher Ausdruck
  - *Anweisung* beliebig z.B.:
    - eine Zuweisung
    - ein Block
    - eine bedingte Anweisung
    - …

```
{ // bedingte Anweisung
  int v=5;
  int summe=0;
  if( v > 0 ) {
     summe += v*v;
     v++;
  }
}
```

Klasse übersetzt - keine Syntaxfehler | gespeichert

©H. Peter Gumm, Philipps-Universität Marburg

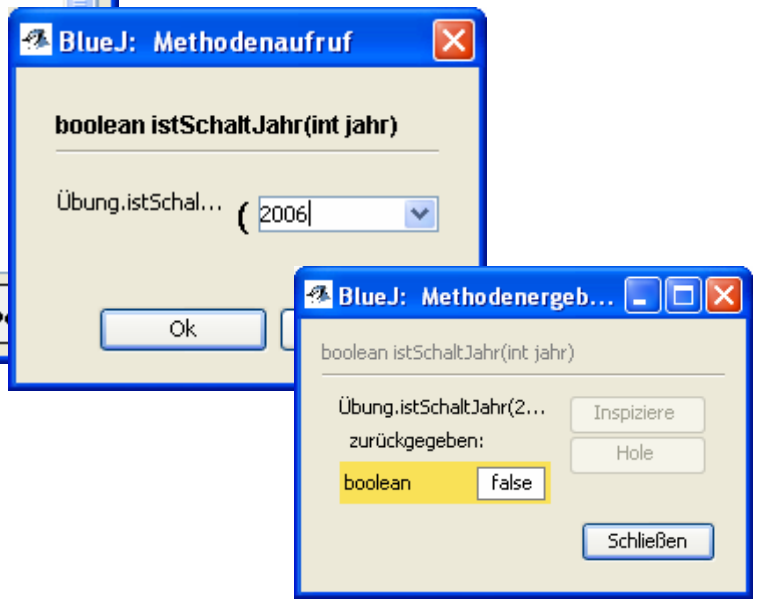
Schachtelung bedingter Anweisungen

Schachtelt man bedingte und Alternativ-Anweisungen, so kann es zu Mehrdeutigkeiten kommen

```
static boolean istSchaltJahr(int jahr )
{ // Was kommt hier raus ?
  if (jahr % 4 == 0)
    if ( jahr % 100 != 0)
      return false;
    if (jahr % 1000 == 0)
      return true;
else return false;
```

Wozu gehört dieses else ?

Klasse übersetzt - keine Syntaxfehler



- Auf welches **`if`** bezieht sich das **`else`** ?
- Regel: ein **`else`** ergänzt stets das letzte (vorhergehende) unergänzte **`if`**.

©H. Peter Gumm, Philipps-Universität Marburg

While-Schleife

- Solange eine Bedingung wahr ist, wird eine Anweisung wiederholt
- Syntax:

`while(` *Bedingung* **`)`** *Anweisung*

- *Bedingung* ein boolescher Ausdruck
- *Anweisung* heißt auch *Körper* oder *Rumpf*

```
static int gauss(int n){
    int summe = 0;
    int k = 1;
    while (k <= 100){
        summe += k;
        k++;
    }
    return summe;
}// end gauss
```

Klasse übersetzt - keine Syntaxfehler | gespeichert

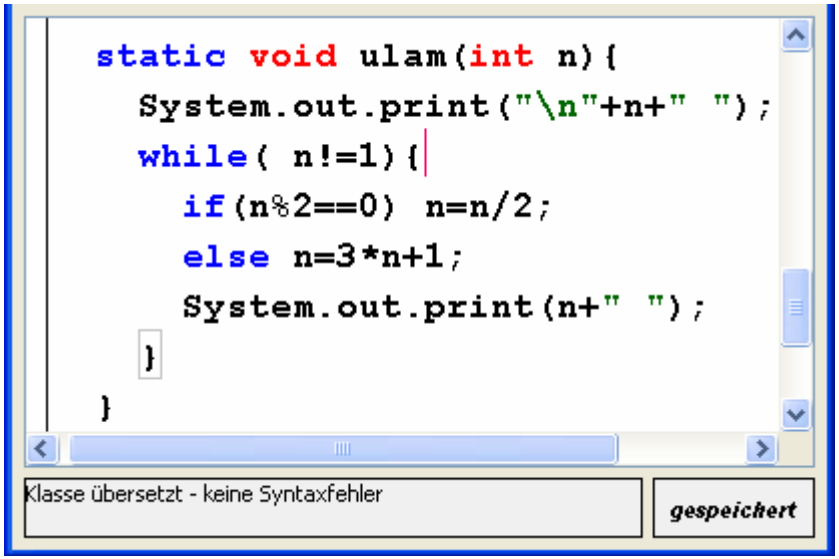
While-Schleifen können
§ 0 mal,
§ endlich oft
§ unendlich oft
Ihren Rumpf ausführen.

Ändern Sie im Beispiel

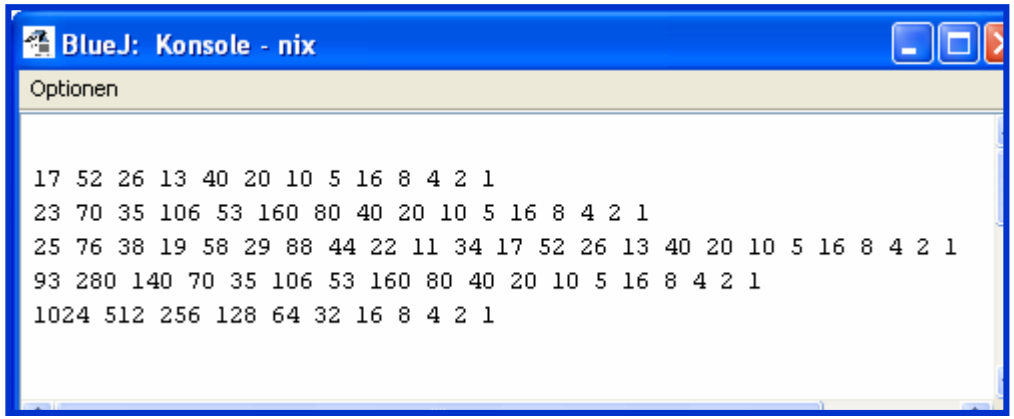
§ `k <= 100` zu `k >= 100`
§ `k <= 100` zu `k < 100`
§ `k++` zu `k--`

©H. Peter Gumm, Philipps-Universität Marburg

Das Ulamsche 3n+1-Problem



- Denken Sie eine Zahl n aus
- Falls
  - n gerade, teile n durch 2
  - sonst multipliziere n mit 3 und addiere 1
- Mit dem Ergebnis E fahre fort wie mit n
- Behauptung: Irgendwann wird n = 1.



Frage:
Kommt da immer 1 raus,
egal mit welchem n>0 ich anfange ?

Für die richtige Antwort gibt es

1.000.000 $

©H. Peter Gumm, Philipps-Universität Marburg

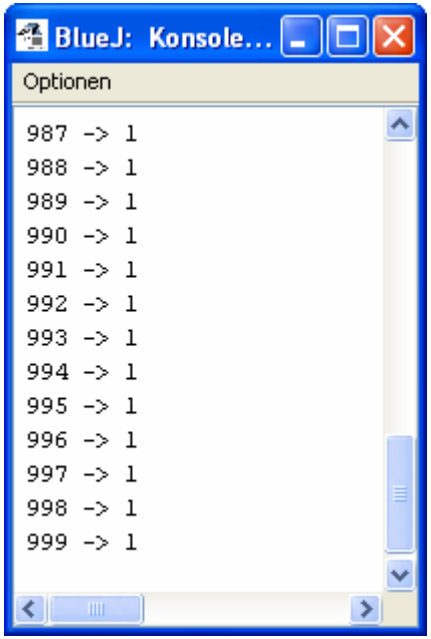
Geschachtelte Schleifen

- Wir wollen feststellen, ob *ulam* für jeden Startwert bei 1 ankommt.
- Beginnend mit n=2 prüfen wir alle Zahlen jede Zahl durch.

while

while

```
static void millionär(){
   int k=2;
   while (k < 1000){
     int n=k;
     while(n!=1){
       if(n%2==0) n=n/2;
       else n=3*n+1;
     }// end-while
     System.out.println(k+" -> 1");
     k++ ;
   }//end-while
}// millionär
```



Verbessern Sie das Programm !

- Muss man überhaupt die geraden Zahlen testen ?
- Welche ungeraden muss man nicht testen ?
- Beschleunigen Sie das Programm mindestens um den Faktor 4

©H. Peter Gumm, Philipps-Universität Marburg

Methoden

- Deklaration

```
Resultattyp Name( Parameterliste )
{ Anweisungen }
```

- Aufruf

```
Name( Parameterwerte)
```

```
System.out.println(gauss(17));

int g = gauss(197);

int n=100;
return gauss(n)- gauss(n-1);
```

4 Aufrufe

```
int gauss(int n)
{
    int summe = 0;
    int k = 1;
    while (k <= 100){
        summe += k;
        k++;
    }
    return summe;
}// end gauss
```


©H. Peter Gumm, Philipps-Universität Marburg

Prozeduren

- Methoden ohne Rückgabewert
  - Ziel ist Effekt zu erzeugen
  - Syntax wie normale Methoden
  - Resultattyp: void

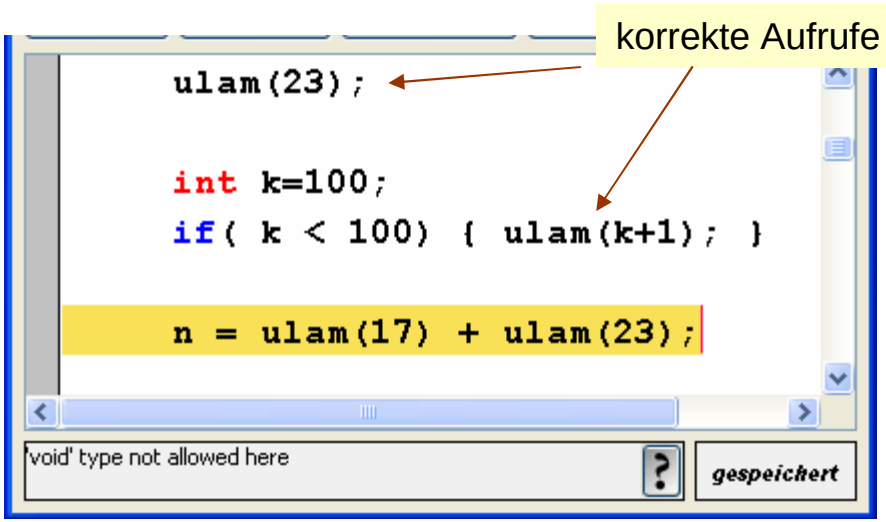
```
void ulam(int n){
  System.out.print("\n"+n+" ");
  while( n!=1){
    if(n%2==0) n=n/2;
    else n=3*n+1;
    System.out.print(n+" ");
  }
}
```

- Deklaration

```
void Name( Parameterliste )
  { Anweisungen }
```

- Aufruf

```
Name( Parameterwerte ) ;
```



©H. Peter Gumm, Philipps-Universität Marburg

return

- Die Anweisung *return*
  - beendet eine Methode sofort
  - liefert den Rückgabewert
- Methoden mit Rückgabewert
  - **müssen** eine Return-Anweisung haben
  - `return kontoStand+16;`
- Methoden ohne Rückgabewert
  - **können** eine Return-Anweisung haben
  - `return ;`

©H. Peter Gumm, Philipps-Universität Marburg

Prozeduren und Methoden

- void-Methoden haben einen Effekt
  - Aufruf einer void-Methode ist *Anweisung*
    - `überweisen(200, meinKonto);`
  - Überall stehen, wo Anweisung möglich ist
    - ```
      if (deinKonto.getKontoStand() > 0)
          deinKonto.überweisen(200,meinKonto);
      ```
- Methoden mit Resultattyp ≠ void liefern einen Wert
  - Aufruf ist *Ausdruck*
    - `getKontostand()`
  - überall wo Ausdruck dieses Typs möglich ist
    - ```
      if(getKontoStand() > 0)
          System.out.println("Keine Miese");
      ```
    - `System.out.println("KontoStand : " + getKontoStand() );`

©H. Peter Gumm, Philipps-Universität Marburg