Informatik I WS 07/08

Tutorium 24

29.11.07

Bastian Molkenthin

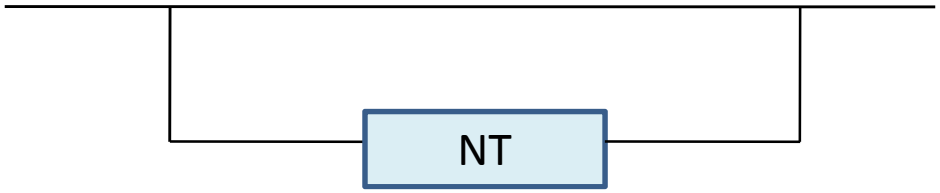
E-Mail: infotut@sunshine2k.de

Web: http://infotut.sunshine2k.de

Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

Review Blatt 4 / Organisatorisches

- EBNF Diagramme korrekt zeichnen!



?

- Eure Regeln müssen zum Diagramm passen!

- Abgabefrist von „Hello World“ bis 7.12 ! Danach 0 Punkte!

Nachtrag Java

Was macht der Ausdruck c = c++; ?

Das gleiche wie c = c!

Grund: Java muss intern das Statement umformen, es sind dann ungefähr so aus:

```
t1 = c;
c = c + 1;
c = t1;
```

1. Auswerten der rechten Seite und merken des Wertes von c im einem Zwischenspeicher.
2. Inkrementiere c.
3. Weise das Ergebnis der rechten Seite (aus Schritt 1) c zu.

Variablen

- Was sind globale Variablen? Was sind lokale Variablen?

  Variablen, die innerhalb einer Methode deklariert werden, heißen *lokal*.

  Variablen, die innerhlab des Klassenrumpfs aber außerhalb einer Methode deklariert werden, heißen *global*.

- Wo leben lokale Variablen?

  Im gesamten Abschnitt, in welchem sie deklariert werden.

- Wo sind lokale Variablen sichtbar?

  Ab dem Punkt der Deklaration bis zum Abschnittsende.

- Wo leben globale Variablen?

  In der gesamten Klasse.

- Wo sind globale Variablen sichtbar?

  In der gesamten Klasse, außer sie wurden überschrieben.

Aufgabe - Lebendigkeit

```
class Test {
 public int globe = 42;

 static int calc (int f, int g){        //2
    int globe;
    globe = max (f, g);
    return f + globe * g;
  }

  static int max (int a, int b)         //3
    if (a > b) return a;
    else return b;
  }


public static void main(String[] args){//1
    int x = 21;
    int y = 1;
    Out.println(calc (x, y));
  }
}
```

Lebendig:
globe(*global*), globe(l*okal*), x, y, f, g

Lebendig:
a, b, globe(*global*), globe(*lokal*), x, y, f, g

Lebendig:
globe (*global*), x,y

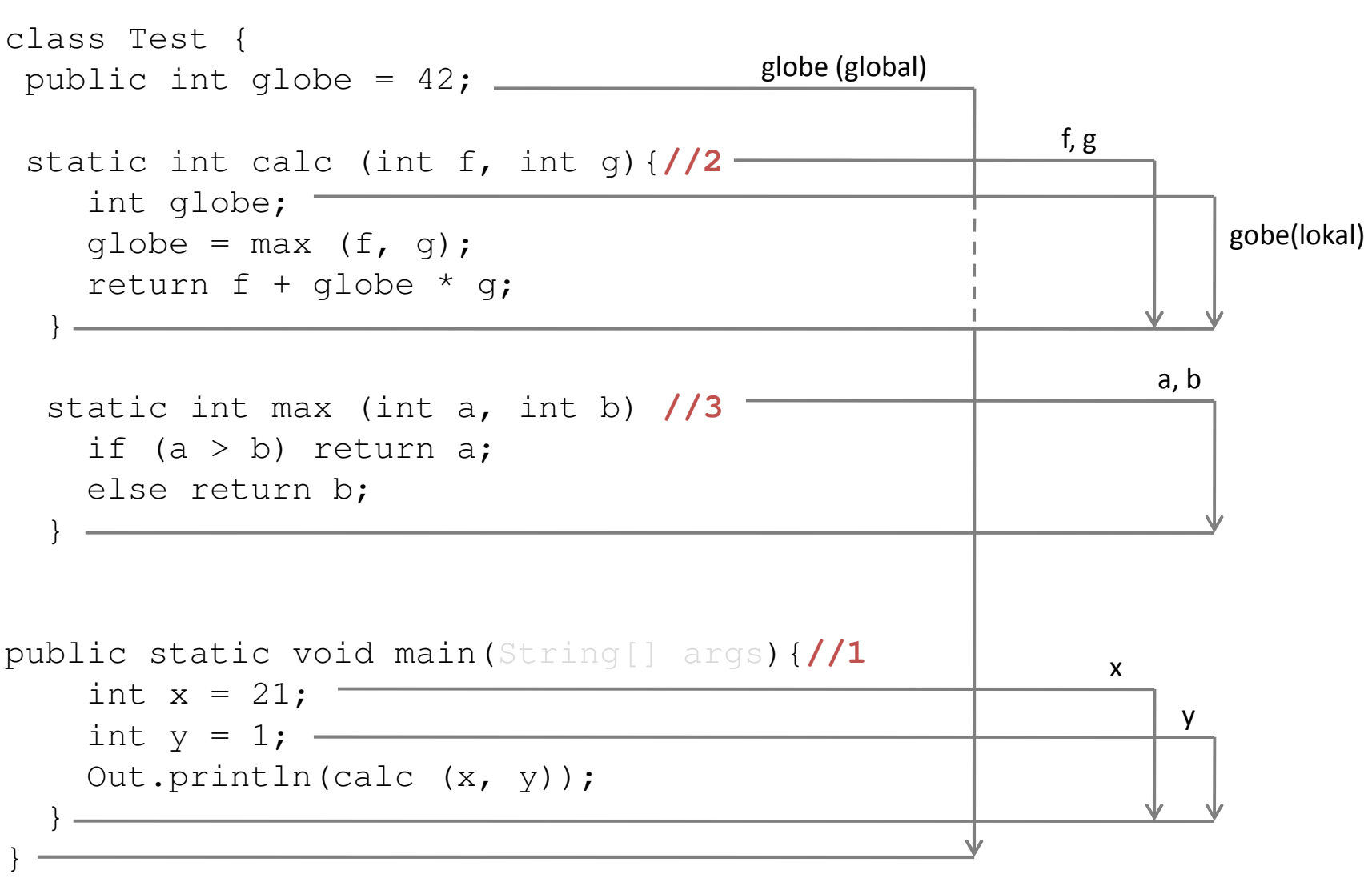
Lebendigkeit Begründung

- In der main-Methode (Stelle 1) leben nur die lokal definierten Variablen *x* und *y* sowie die globale Variable *globe*.

- Da die calc-Methode von der main-Methode aufgerufen wird, leben an Stelle 2 neben den lokalen Variablen f und g auch alle Variablen der main-Methode (*x*, *y* und *globe*).

- Da die max-Methode von der calc-Methode aufgerufen wird, leben an Stelle 3 neben den in der Methode selbst definierten Variablen *a* und *b* auch alle Variablen der calc-Methode (und somit auch der main-Methode).

Aufgabe - Sichtbarkeit



Lösung:

1: sichtbar ist globe (global)
2: sichtbar sind f,g, globe (global)
3 : sichtbar sind a,b, globe (global)

Aufgabe 2 - Lebendigkeit

Lösung

```
class Program{
    static int offset = 3;
        public static void main (String[] arg) {
            int offset = In.readInt();          /* 1 */
            int in1 = In.readInt();
            int in2 = In.readInt();
           Out.println(foobar(offset,in1,in2));
        }

        public static int foobar(int add, int a,int b) {
            int min = calcMin(a,b);             /* 2 */
            min  = doSomething(min) + add;
            return min;
        }

        public static int doSomething(int some) {
            int lala = some + offset;           /* 3 */
            return lala;
        }

} // end class Program
```

offset (global), offset (lokal), in1, in2

add, a, b, min, Stelle 1

some, lala, Stelle 2

Aufgabe 2 - Sichtbarkeit

```
class Program{
    static int offset = 3;
        public static void main (String[] arg) {
            int offset = In.readInt();          /* 1 */
            int in1 = In.readInt();
            int in2 = In.readInt();
           Out.println(foobar(offset,in1,in2));
        }

        public static int foobar(int add, int a,int b) {
            int min = calcMin(a,b);             /* 2 */
            min  = doSomething(min) + add;
            return min;
        }

        public static int doSomething(int some) {
            int lala = some + offset;           /* 3 */
            return lala;
        }
}
```


Lösung:

1: sichtbar ist offset (lokal)
2: sichtbar sind a, b, add, offset (global)
3 : sichtbar sind some, lala, offset(global)

Schleifen

Eine Schleife erfüllt den Zweck bestimmte Operationen häufig hintereinander ausführen, bis eine Abbruchbedingung erreicht ist

In Java gibt es die drei wichtigen Schleifentypen:

- *For-Schleife*
- *While-Schleife*
- *Do-While-Schleife*

Meistens lassen sich die verschiedenen Schleifen ineinander überführen, aber in der Regel drängt sich ein bestimmter Typ auf

For - Schleife

- Für den Fall, dass etwas mit einer bestimmten Anzahl von Wiederholungen ausgeführt werden soll.
- Im Schleifenkopf wird ein Zähler integriert

Syntax:

```
for (Initialisierung; Bedingung; Inkrementierung) { /*Code*/ }
```

Beispiel:

```
int sum = 0;
for (int i = 0; i < 10; i++) {
        sum += i;
}
```

Berechnet 0+1+2+3+4+5+6+7+8+9 = 45

While - Schleife

- Etwas so oft ausführen, bis eine/mehrere Bedingung(en) erfüllt sind
- Kopfgesteuert

Syntax:

```
while (Bedingung){ /*Code*/ }
```

Beispiel:

```
while (tired == false || z < 10) {
        // Diverse Anweisungen
        z--;
}
```

Do-While - Schleife

- Etwas so oft ausführen, bis eine/mehrere Bedingung(en) erfüllt sind
- Wird mindestens einmal durchlaufen
- Fußgesteuert

Syntax:

```
do {
/*Code*/
} while (Bedingung);
```

Beispiel:

```
do {
        // Diverse Anweisungen
        z--;
} while (tired == false || z < 10);
```

Schleifen - break

Wird innerhalb einer *for*-, *while*- oder *do/while*-Schleife eine break-Anweisung eingesetzt, so wird der Schleifendurchlauf beendet und die Abarbeitung bei der ersten Anweisung nach der Schleife fortgeführt.

Beispiel:

```
int i = 0;
while (true) {
    if (++i == 3)
        break;
    Out.println("Durchlauf " + i);
}
```

Ausgabe:

```
Durchlauf 1
Durchlauf 2
```

Schleifen - continue

Innerhalb einer *for-*, *while-* oder *do/while*-Schleife lässt sich eine continue-Anweisung einsetzen, die zum Schleifenkopf zurückgeht.

Beispiel:

```
for (int i = 0; i < 8; i++)
{
    if (i % 2 == 0)
        continue;
    Out.println("i ist " + i);
}
```

Ausgabe:

```
i ist 1
i ist 3
i ist 5
i ist 7
```

Anmerkung: Break und Continue sollten möglichst vermieden werden!

Aufgabe - Schleifenwahl

Welches ist der geeignetste Schleifentyp und warum?

Aufgabe: Zunächst ist ein positiver ganzzahliger Grenzwert einzulesen. Danach werden ganze Zahlen eingelesen und nach Eingabe jeder Zahl wird die Summe der bisher gelesenen Zahlen ausgegeben, bis die Summe (inklusive der zuletzt eingegebenen Zahl) den Grenzwert überschritten hat.

Lösung: *do-while-Schleife*, da die Abbruchbedingung erst nach der Summation und Ausgabe überprüft werden muss.

Aufgabe - Schleifenwahl

Welches ist der geeignetste Schleifentyp und warum?

Aufgabe: Bis zur Eingabe einer 0 sind ganze Zahlen einzulesen. Sofern es sich bei der Eingabe um keine 0 handelt, wird nach Eingabe die Summe der bisher gelesenen Zahlen ausgegeben.

Lösung: *while-Schleife*, da die Abbruchbedingung bereits vor der ersten Summation und Ausgabe überprüft werden muss.

Aufgabe - Schleifentransformation

Wandelt folgende for-Schleife in eine while und eine do-while Schleife um.
Verwende dabei keine break- oder continue-Anweisung!

```
int s = 0;

for (;;) {
        int x = In.readInt();
        if (x<0) break;
        s = s + x;
}
```

Lösung (while-Schleife):

```
int s = 0;
int x = In.readInt();
while (x >= 0) {
        s = s + x;
        x = In.readInt();
}
```

Aufgabe - Schleifentransformation (2)

Wandelt folgende for-Schleife in eine while und eine do-while Schleife um.
Verwende dabei keine break- oder continue-Anweisung!

```
int s = 0;

for (;;) {
        int x = ln.readInt();
        if (x<0) break;
        s = s + x;
}
```

Lösung (do-while-Schleife):

```
int s = 0;
int x = 0;
do {
        s = s + x;
        x = ln.readInt();
} while (x >= 0);
```

Call By Value

Call-by-value bei einer Paramterübergabe bedeutet, dass bei der Verwendung einer Variablen als Parameter der Inhalt einer Variable (also eine Kopie) an eine Methode übergeben wird.

Beispiel:

```
public static void increase (int x) {
    x = x + 1;
}
public static void main(String args[]) {
    int a = 5;
    increase(a);
    Out.println(a);        // Ausgabe ist 5!
}
```

Java verwendet für primitive Datentypen das call-by-value Prinzip!

Call By Reference

Call-by-reference bei einer Paramterübergabe bedeutet, dass bei der Verwendung einer Variablen als Parameter eine Referenz (also ein Zeiger) auf eine Variable an eine Methode übergeben wird.

Falls somit die Methode die Variable modifiziert, ändert sich die Variable auch aus Sicht des Aufrufers.

Objekte wie Klassen werden per Referenz übergeben.

Char / Unicode

- Der Datentyp *char* nimmt einzelnes Unicode-Zeichen auf, ist somit 16Bit groß.

  Bsp:
  ```
  char c = 'g';
  ```

- char ist unsigned (kein +/-)

- Die unteren 128 Zeichen ensprechen in Java dem ASCII

- Der Zeichenwert wird intern als Unicode gespeichert. Um diesen auszugeben, kann man die char-Variable zu int casten.

  Bsp:
  ```
  char c = 'g';
  Out.println("Unicode value of g is " + (int)c);
  ```

  Ausgabe: "Unicode value of g is 103".

- Falls man nicht castet , wird das Zeichen ausgeben.

  Bsp:
  ```
  char c = 'g';
  Out.println("Value is " + c);
  ```

  Ausgabe: "Value is g".

Arrays

- Array ist ein spezieller Datentyp, der mehrere Elemente zu einer Einheit zusammenfasst.
- Alle Elemente haben den gleichen Typ.
- Zugriff auf Elemente über ganzzahligen Index, der Index fängt immer bei 0 an!
- Deklaration: `int[] field;`

  Reserviert noch kein Speicherplatz für die Elemente!

Mit **new** Operator!

```
int[] field = new int[20];
```

Erzeugt Array mit Speicherplatz für 20 int-Werte.

- Länge eines Arrays abfragbar mit length. (keine Funktion, sondern Variable!)

```
int lenOfField = field.length;
```

(Mehrdimensionale) Arrays

- Arrays könne auch mit Werrten initialisert werden.

```
int[] mynumbers = { 4, 7, 23, 45, 13 };
```

Mehrere Schreibweisen möglich!

- Mehrdimensionale Arrays sind „Arrays von Arrays“.

```
int[][] matrix3x3 = new int[3][3];
```

Zeilen Spalten

```
a[0][0]  a[0][1]  a[0][2]
a[1][0]  a[1][1]  a[1][2]
a[2][0]  a[2][1]  a[2][2]
```

- Mehrdimensionale Arrays müssen nicht reckeckig sein!

```
int[][] m = new int[ 3 ][];
for ( int i = 0; i < 3; i++ )
     m[ i ] = new int[ i + 1 ];
```

Erzeugt dreieckiges Array mit Zeilen der Länge 1, 2 und 3.

Klammern [ ] vor oder hinter den Variablennamen ?

- Klammern [] sind vor und hinter dem Variablennamen möglich!

```
int[] prims;        int prims[];
```

sind beide möglich!

- Kleiner Unterschied, wenn mehrere Variablen deklariert werden:

```
int[] prims, matrix[], threeDimMatrix[][]
```

entspricht

```
int prims[], matrix[][], threeDimMatrix[][][]
```

z.B.

```
int []prim, i;
i = 2;
```

→ Fehler! **i** ist Feld!

- Doch vielleicht war beabsichtigt, i als int-Variable zu deklarieren!

```
int prim[], i;
i = 2;
```

→ OK!

Aufgabe

Fülle ein nichtrechteckiges Array mit Werten des Pascal‘schen Dreiecks!

```
1
1 1
1 2  1
1 3  3  1
1 4  6  4 1
1 5 10 10 5  1
1 6 15 20 15 6 1
```

Das Dreieck ist so aufgebaut, dass die Elemente unter einer Zahl genau die Summe der beiden direkt darüber stehenden Zahlen bilden. Die Ränder sind mit Einsen belegt.

In jeder Ebene soll ein Array mit passender Länge dynamisch erzeugt werden.

Lösung

```
int[][] triangle = new int[7][];

for ( int i = 0; i < triangle.length; i++ )
{
    triangle[i] = new int[i + 1];

    for ( int j = 0; j <= i; j++ )
    {
        if ( (j == 0) || (j == i) )
            triangle[i][j] = 1;
        else
            triangle[i][j] = triangle[i - 1][j - 1] + triangle[i - 1][j];

         Out.print(triangle[i][j] + " ");
    }
    Out.println();
}
```

Das wars für heute…

Fragen ???

Viel Spaß mit dem Übungsblatt!