Übersicht

Inhalte

- ✓ Blöcke und ihre Besonderheiten
- Methodendefinition und -aufruf
- Polymorphie von Operationen
- Überladen von Methoden
- Parameterliste variabler Länge
- Parameterübergabe beim Programmaufruf
- Iteration und Rekursion

Blöcke und ihre Besonderheiten I

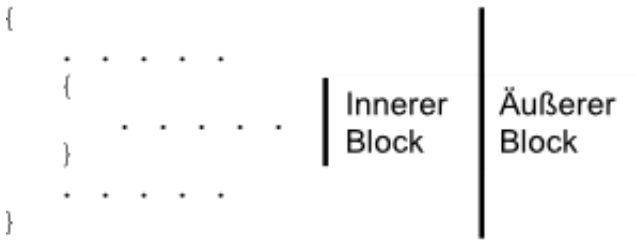
Ein Block benötigt man aus zwei Gründen:

- Zum einen ist der Rumpf einer Methode ein Block
- Zum anderen gilt ein Block syntaktisch als eine einzige Anweisung. Daher kann ein Block auch da stehen, wo von der Syntax her nur eine einzige Anweisung zugelassen ist, wie z.B. im *if*-oder *else*-Zweig einer *if-else*-Anweisung.
- Hinter einem Block kommt ***kein*** Strichpunkt
- Ist eine Folge von Anweisungen, die sequenziell hintereinander ausgeführt wird

```
{
    . . . . .
    Deklarationsanweisungen
    . . . . .
    Anweisungen
    . . . . .
    Deklarationsanweisungen
    . . . . .
    Anweisungen
    . . . . .
}
```

Blöcke und ihre Besonderheiten II

- Ein Block zählt syntaktisch als eine einzige Anweisung. Im Gegensatz zu einer normalen Anweisung besteht bei einem Block jedoch die Möglichkeit, **Block-lokale Variable** einzuführen.



(Schachtelung von Blöcken)

- **In einem inneren Block definierte Variable** sind nur innerhalb dieses Blockes **sichtbar, in einem umfassenden Block** sind sie **unsichtbar**. Variable, die in einem umfassenden Block definiert sind, sind für einen folgenden inneren Block **auch sichtbar.**
- Bei **Java sind identische Namen im inneren und äußeren Block nicht zugelassen.** Es resultiert ein Kompilierfehler.

Blöcke und ihre Besonderheiten III

Gültigkeit, Sichtbarkeit und Lebensdauer

- Die **Lebensdauer** ist die Zeitspanne, in der die virtuelle Maschine der Variablen einen **Platz im Speicher** zur Verfügung stellt. Mit anderen Worten, während ihrer Lebensdauer besitzt eine Variable einen Speicherplatz

- Die **Gültigkeit** einer Variablen bedeutet, dass an einer Programmstelle der Namen einer Variablen dem Compiler durch eine Vereinbarung bekannt ist.

- Die **Sichtbarkeit** einer Variablen bedeutet, dass man von einer Programmstelle aus die Variable sieht, das heißt, dass man auf sie über ihren Namen zugreifen kann.

Blöcke und ihre Besonderheiten IV

- Sichtbarkeit von Variablen in Blöcken

```
public class BlockTest
{
   public void zugriff(){

      int aussen = 7;
      if (aussen == 7){

         int innen = 8;
         System.out.print ("Zugriff auf die Variable");
         System.out.println (" des aeusseren Blocks: " + aussen);
         System.out.print ("Zugriff auf die Variable");
         System.out.println (" des inneren Blocks: " + innen);
      }
   }

   public static void main (String[] args){
      BlockTest ref = new BlockTest();
      ref.zugriff();
   }
}
```

```
Zugriff auf die Variable des aeusseren Blocks: 7
Zugriff auf die Variable des inneren Blocks: 8
```

Blöcke und ihre Besonderheiten V

Variable	Sichtbarkeits- und Gültigkeitsbereich	Lebensdauer
Lokal	im Block einschließlich inneren Blöcken	Block ab Definition
Instanzvariable	im Objekt selbst	vom Anlegen des Objektes bis das Objekt nicht mehr referenziert wird
Klassenvariable	in allen Objekten der entsprechenden Klasse und in allen zugehörigen Klassenmethoden	vom Laden der Klasse bis die Klasse nicht mehr benötigt wird

- Bei lokalen Variablen fallen Gültigkeit und Sichtbarkeit zusammen. Bei Datenfeldern muss man prinzipiell zwischen **Gültigkeit** und **Sichtbarkeit** unterscheiden.

Blöcke und ihre Besonderheiten VI

- Sichtbarkeit von Variablen in Blöcken

```
public class Sichtbar{
   private int x = 0;          // Datenfeld x

   public void zugriff(){
      int x = 7;               // lokale Variable x

      // Ausgabe der lokalen Variablen x
      System.out.println ("Lokale Variable x: " + x);

      // this zeigt auf das aktuelle Objekt und damit ist this.x die
      // x-Komponente des aktuellen Objektes
      // Ausgabe des Datenfeldes x
      System.out.println ("Datenfeld x: " + this.x);
   }

   public static void main (String[] args){
      Sichtbar sicht = new Sichtbar();
      sicht.zugriff();
   }
}
```

```
Lokale Variable x: 7
Datenfeld x: 0
```

Übersicht

Inhalte

- ✓ Blöcke und ihre Besonderheiten
- ✓ Methodendefinition und -aufruf
- Polymorphie von Operationen
- Überladen von Methoden
- Parameterliste variabler Länge
- Parameterübergabe beim Programmaufruf
- Iteration und Rekursion

Methodendefinition und -aufruf I

- Die **Methodendeklaration** sieht im allgemeinen Fall folgendermaßen aus:

Modifikatoren Rückgabetyp Methodenname (Typ1 formalerParameter1,
Typ2 formalerParameter2,
……..
TypN formalerParameterN,)

- Parameterlose Methode:
  - Leeres Klammerpaar statt Liste formaler Parameter

- Rückgabewert mittels ***return***-Anweisung
- Wird das Schlüsselwort ***void*** statt eines Rückgabetyps angegeben, so ist kein *return* notwendig. Es kann aber jeder Zeit mir ***return*** die Abarbeitung der Methode abgebrochen werden. Damit wird ein sofortiger Rücksprung zur Aufrufstelle bewirkt. In diesem Fall darf mit der ***return*** Anweisung kein Wert zurückgegeben werden.

Methodendefinition und -aufruf I

- Methodenkopf / Methodenrumpf

```
Methodendeklaration    // Methodenkopf
{                      //
                       // Methodenrumpf
}                      //
```

- Methodendeklaration

```
Modifikatoren Rückgabetyp Methodenname (Typ1 Parameter1,
                                        Typ2 Parameter2,
                                        …,
                                        TypN ParameterN)
```

```
// Beispiel
static int berechneSumme (int zahl1, int zahl2)
```

- Rückgabewert

```
int getX(){
    return x;
}
```

Methodendefinition und -aufruf II

- Eine Methode kann mit ***return*** nur **einen einzigen Wert** zurückgeben. Möchte man mehrere Werte zurückgeben, so kann dies über Referenzen auf Objekte in der Parameterliste gehen oder über die Schaffung eines Objektes mit mehreren Datenfeldern, auf das mit *return* eine Referenz zurückgegeben wird.

- An der aufrufenden Stelle darf der Wert, den eine Methode liefert, ignoriert werden. Mit anderen Worten, man kann eine Methode, die einen Rückgabewert hat, einfach aufrufen, ohne den Rückgabewert abzuholen.

Methodendefinition und -aufruf III

Formale und aktuelle Parameter

- Mit den **formalen Parametern** wird festgelegt, **wie viele Übergabeparameter** existieren, **von welchem Typ** diese sind **und welche Reihenfolge** sie haben.
- Beim Aufruf werden den formalen Parametern die Werte der **aktuellen Parameter** zugewiesen.
- Beim Aufruf einer Methode mit Parametern finden Zuweisungen statt. Der **Wert** eines **aktuellen Parameter** wird dem entsprechenden **formalen Parameter zugewiesen.** Eine solche Aufrufschnittstelle wird als **call by value-Schnittstelle** bezeichnet.

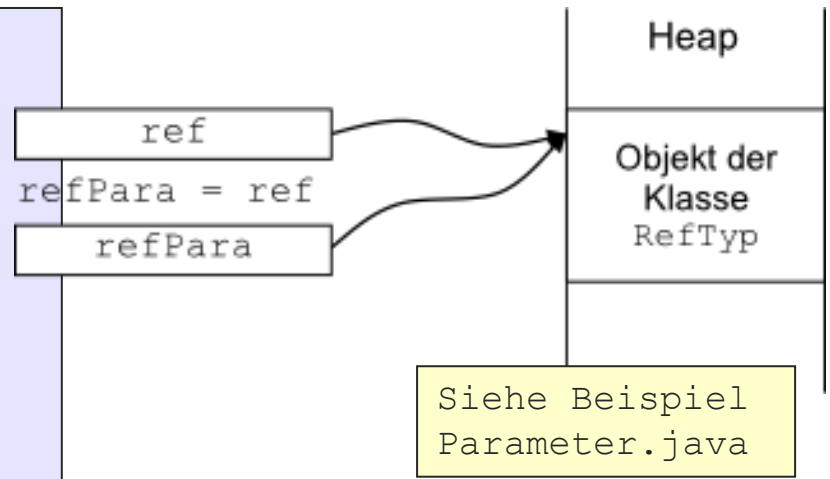
```
…
ref.setX(20);
…

void setX(int newX){
    this.x = newX;
}
```

Methodendefinition und -aufruf IV

- Parameterübergabe von Referenzen *(Der formale Parameter referenziert dasselbe Objekt wie der aktuelle Parameter)*
- Werden Referenzen übergeben, so referenziert der formale Parameter dasselbe Objekt wie der aktuelle Parameter. Eine **Operation auf dem formalen Referenzparameter erfolgt** auf dem Objekt, auf das die Referenz zeigt, in anderen Worten **auf dem referenzierten Objekt**.

```
class RefTyp{
    public int x;
}
…
public static void methode (RefTyp refPara){
    refPara.x = 2;
}
…
RefTyp ref = new RefTyp();
ref.x = 1;
methode(ref);
System.out.println ("Wert von x ist "+ref.x);
```




```
Wert von x ist 2.
```

Aufgaben zu Blöcken und Methoden

Aufgabe 06.01 Gegeben sei folgende Klasse:

```
public class Aufg_06_01 {
    public static void main (String[] args){
        int[] array = {4, 19, 20, 7, 36, 18, 1, 5};
        IntArray intArray = new IntArray(array);

        // Anzeigen
        intArray.print();

        // Max anzeigen
        System.out.println("Max: "+intArray.max());

        // Min anzeigen
        System.out.println("Min: "+intArray.min());

        // Mittelwert anzeigen
        System.out.println("Mittelwert: "+intArray.average());

        // sortieren
        intArray.sort();

        // Anzeigen
        intArray.print();
    }
}
```

Aufgaben zu Blöcken und Methoden

Aufgabe 06.01 Fortsetzung

Implementieren Sie die Klasse IntArray mit allen notwendigen Methoden:

- Konstruktor: erhält ein int-Array als Parameter
- public float average(): gibt den Mittelwert zurück
- public int max(): gibt den größten Wert zurück
- public int min(): gibt den kleinsten Wert zurück
- public void print(): Ausgabe des Array-Inhalts
- public void sort(): sortiert das Array mit Arrays.sort(…)
- Video: Bubble-Sort

Testen Sie Ihre Implementierung, indem Sie die die Klasse Aufg_06_01 starten.

Aufgaben zu Blöcken und Methoden

Aufgabe 06.01 Fortsetzung

Verwenden Sie zum Sortieren des Arrays Bubble-Sort

Bekannter Sortieralgorithmus: die größten Elemente steigen wie Luftblasen nach oben.

1) es gibt eine äußere Schleife, die sooft durchlaufen wird, wie die Liste Elemente hat
2) es gibt eine innere Schleife, in der die Elemente verglichen und evtl. getauscht werden

    2.1) Schauen, ob das Element mit Index i größer ist, als das mit Index i+1
        wenn ja, dann Elemente tauschen und Index um 1 erhöhen u.s.w.
    2.2) nach jedem Durchlauf der inneren Schleife steht das größte Element ganz rechts, im nächsten Durchlauf muss daher nur noch eine um ein Element kürzere Liste sortiert werden

Übersicht

Inhalte

- ✓ Blöcke und ihre Besonderheiten
- ✓ Methodendefinition und -aufruf
- ✓ Polymorphie von Operationen

Überladen von Methoden

Parameterliste variabler Länge

Parameterübergabe beim Programmaufruf

Iteration und Rekursion

Polymorphie von Operationen I

- Eine Klasse stellt einen **Namensraum** dar. Damit ist es möglich, dass verschiedene Klassen dieselbe Operation implementieren, in anderen Worten, derselbe Methodenkopf kann in verschiedenen Klassen auftreten.
- Je nach Klasse kann eine Operation in verschiedenen Implementierungen – sprich in verschiedener Gestalt – auftreten. Man spricht hierbei auch von der **Vielgestaltigkeit (Polymorphie) von Operationen.**

```java
// Datei: Bruch2.java

public class Bruch2
{
   private int zaehler;
   private int nenner;

   public Bruch2 (int zaehler, int nenner)
   {
      this.zaehler = zaehler;
      this.nenner = nenner;
   }

   public void print()
   {
      System.out.print ("Der Wert des Quotienten von " + zaehler);
      System.out.print (" und " + nenner + " ist " + zaehler
                        + " / ");
      System.out.println (nenner);
   }
}
```

```java
// Datei: Person2.java

public class Person2
{
   private String name;
   private String vorname;
   private int alter;

   // Konstruktur fuer die Initialisierung der Datenfelder
   public Person2 (String name, String vorname, int alter)
   {
      this.name = name;
      this.vorname = vorname;
      this.alter = alter;
   }

   public void print()
   {
      System.out.println ("Name    : " + name);
      System.out.println ("Vorname : " + vorname);
      System.out.println ("Alter   : " + alter);
   }
}
```

Polymorphie von Operationen II

- Jedes Objekt trägt die Typinformation, von welcher Klasse es ist, immer bei sich.
- Daher ist ein Methodenaufruf immer an das spezifische Objekt (bzw. bei statischen Methoden an die Klasse) gebunden

```
1  // Datei: Polymorphie.java
2
3  public class Polymorphie
4  {
5     public static void main (String[] args)
6     {
7        Bruch2 b;
8        b = new Bruch2 (1, 2);
9        b.print();
10
11       Person2 p;
12       p = new Person2 ("Mueller", "Fritz", 35);
13       p.print();
14    }
15 }
```

```
1/2
Name    :Mueller
Vorname:Fritz
```

Übersicht

Inhalte

- ✓ Blöcke und ihre Besonderheiten
- ✓ Methodendefinition und -aufruf
- ✓ Polymorphie von Operationen
- ✓ Überladen von Methoden

Parameterliste variabler Länge

Parameterübergabe beim Programmaufruf

Iteration und Rekursion

Überladen von Methoden I

- Ein **Überladen** erfolgt durch die Definition verschiedener Methoden mit **gleichem Methodennamen,** aber **verschiedenen Parameterlisten**. Der Aufruf der richtigen Methode ist Aufgabe des Compilers.
- Die Signatur setzt sich zusammen aus dem Methodennamen und der Parameterliste und muss eindeutig sein!

> **Signatur = Methodenname + Parameterliste**
>
> Der Rückgabetyp ist **nicht** Bestandteil der Signatur!

Achtung:

- Es ist nicht möglich, in der gleichen Klasse zwei Methoden mit gleichem Methodennamen und gleicher Parameterliste (gleicher Signatur), aber verschiedenen Rückgabetypen zu vereinbaren.
- Wenn keine exakte Übereinstimmung gefunden wird, wird vom Compiler versucht, die spezifischste Methode zu finden. Besser ist jedoch stets, selbst für passende aktuelle Parameter zu sorgen, gegebenfalls durch eine explizite Typkonvertierung

Überladen von Methoden II

- Beispiel der Methode abs() aus der Klasse java.lang.Math zur Ermittlung des Betrags eines arithmetischen Ausdrucks

```
public static int      abs (int)

public static float    abs (float)

public static long     abs (long)

public static double   abs (double)
```

Überladen von Methoden III

- Beispiel Parser

```
public class Parser{
   // Wandelt den String var in einen int-Wert.
   public static int parseInt (String var){
      return Integer.parseInt (var);
   }

   // Wandelt den Stringanteil von der Position pos
   // bis zum Stringende in einen int-Wert.
   public static int parseInt (String var, int pos){
      var = var.substring (pos);
      return Integer.parseInt (var);
   }

   // Wandelt den Stringanteil von der Position von bis
   // zur Position bis in einen int-Wert.
   public static int parseInt (String var, int von, int bis){
      var = var.substring (von, bis);
      return Integer.parseInt (var);
   }
}
```

Überladen von Methoden IV

- Beispiel Parser Fortsetzung

```
public class TestParser{

   public static void main (String[] args){

      String[] daten ={"Rainer Brang","Hauptstr. 17", "73732 Esslingen","25"};
      System.out.println ("Alter: " + Parser.parseInt (daten [3]));
      System.out.println ("Hausnummer: " +Parser.parseInt (daten [1], 10));
      System.out.println ("Postleitzahl: " +Parser.parseInt (daten [2], 0,5));
   }
}
```

```
Alter: 25
Hausnummer: 17
Postleitzahl: 73732
```

Übersicht

Inhalte

- ✓ Blöcke und ihre Besonderheiten
- ✓ Methodendefinition und -aufruf
- ✓ Polymorphie von Operationen
- ✓ Überladen von Methoden
- ✓ Parameterliste variabler Länge

Parameterübergabe beim Programmaufruf

Iteration und Rekursion

Parameterliste variabler Länge

- Die variable Parameterliste muss immer am Ende der Parameterliste stehen.
- Typ innerhalb der Liste muss gleich sein

```
1  // Datei: TestVarargs.java
2
3  public class TestVarargs
4  {
5     public static void main (String[] args)
6     {
7        varPar (1, 2, 3, "Dies", "ist", "ein", "Test!");
8     }
9
10    public static void varPar (int a, int b, int c, String... str)
11    {
12       System.out.printf ("Erster Parameter: %d\n", a);
13       System.out.printf ("Zweiter Parameter: %d\n", b);
14       System.out.printf ("Dritter Parameter: %d\n", c);
15
16       for (String element : str)
17       {
18          System.out.println ("Variabler Anteil: " + element);
19       }
20    }
21 }
22
```

Übersicht

Inhalte

- ✓ Blöcke und ihre Besonderheiten
- ✓ Methodendefinition und -aufruf
- ✓ Polymorphie von Operationen
- ✓ Überladen von Methoden
- ✓ Parameterliste variabler Länge
- ✓ Parameterübergabe beim Programmaufruf

Iteration und Rekursion

Parameterübergabe beim Programmaufruf I

- Der *main*-Methode kann ein *String-Array* übergeben werden
- Die Parameter werden Leerzeichen-getrennt hinter den Aufruf geschrieben

```
1  // Datei: StringTest.java
2
3  public class StringTest
4  {
5     public static void main (String[] args)
6     {
7        String a = "Java";
8        String b = args [0];
9        if (a.equals (b))
10       {
11          System.out.println ("Der String war Java");
12       }
13       else
14       {
15          System.out.println ("Der String war nicht Java");
16       }
17    }
18 }
19
```

Parameterübergabe beim Programmaufruf II

- Beispiel mit zwei Zahlen

```
java AddInteger 5 4
```

```
public class AddInteger {

   public static void main (String[] args){

      if (args.length != 2){
         System.out.println ("FEHLER: Falsche Parameteranzahl");
         System.out.println ("Bitte zwei Parameter eingeben");
         System.out.println ("AddInteger <int1> <int2>");

      }else{
         int i1 = Integer.parseInt (args [0]);
         int i2 = Integer.parseInt (args [1]);
         System.out.println (args [0]+" + "+args [1]+" = "+(i1+i2));
      }
   }
}
```

```
5 + 4 = 9
```

Übersicht

Inhalte

- ✓ Blöcke und ihre Besonderheiten
- ✓ Methodendefinition und -aufruf
- ✓ Polymorphie von Operationen
- ✓ Überladen von Methoden
- ✓ Parameterliste variabler Länge
- ✓ Parameterübergabe beim Programmaufruf
- ✓ Iteration und Rekursion

Iteration und Rekursion

- Ein Algorithmus heißt **iterativ**, wenn bestimmt Abschnitte des Algorithmus innerhalb einer einzigen Ausführung des Algorithmus mehrfach durchlaufen werden. Er heißt **rekursiv**, wenn er Abschnitte enthält, die sich selbst direkt oder indirekt aufrufen.

Rekursive Methoden:

- Funktionieren analog zu den rekursiven Funktionen in C
- Methoden, die sich direkt oder indirekt aufrufen

- Performance
  - Iterative Methoden sind aus Performance-Gründen den rekursiven Lösungen vorzuziehen

Aufgaben

Aufgabe 06.02

Analysieren Sie das folgende Programm. Was erwarten Sie als Ausgabe?

```
// Datei: SichtbarAufg.java

public class SichtbarAufg
{
   private int wert = 7;

   public int zugriff()
   {
      int wert = 77;
      return wert;
   }

   public static void main (String [] args)
   {
      SichtbarAufg sich = new SichtbarAufg();
      System.out.println (sich.zugriff());
   }
}
```

Aufgaben

Aufgabe 06.03

Analysieren Sie das folgende Programm. Was erwarten Sie als Ausgabe?

```
// Datei: GueltigkeitAufg.java

public class GueltigkeitAufg
{
   private int wert = 7;

   public int zugriff()
   {
      int tmp = wert;
      int wert = 77;
      return tmp;
   }

   public static void main (String [] args)
   {
      GueltigkeitAufg guelt = new GueltigkeitAufg();
      System.out.println (guelt.zugriff());
   }
}
```

Aufgaben

Aufgabe 06.04 - Sichtbarkeit und Verdecken von Instanzvariablen

Ausgangspunkt ist das Programm aus Aufgabe 06.02. Wie muss die zugriff()-Methode verändert werden, damit sie nicht den Wert der lokalen Variablen, sondern der Instanzvariablen zurückgibt?

Die lokale Variable soll nicht umbenannt oder entfernt werden!

Ergänzen Sie das Programm.

Aufgaben

Aufgabe 06.05 - Polymorphie

Berechnung von Flächeninhalten

Die Flächeninhalte von Quadraten und Kreisen werden unterschiedlich berechnet. Ergänzen Sie das Programmfragment ***PolymorpheOperation.java*** aus dem Online Campus um die polymorphe Operation "berechneFlaecheninhalt", sodass der jeweils korrekte Flächeninhalt ermittelt wird. Fehlende Stellen sind durch
HIER ERGÄNZEN markiert.

Aufgabe 06.06 – Überladen von Methoden

Überladene Methode zur Summenberechnung

Das Programmfragment ***UeberladeneMethoden.java*** aus dem Online Campus stellt einen minimalistischen Taschenrechner dar. Überladen Sie die Methode zur Summenberechnung, damit auch Kommazahlen addiert werden können.

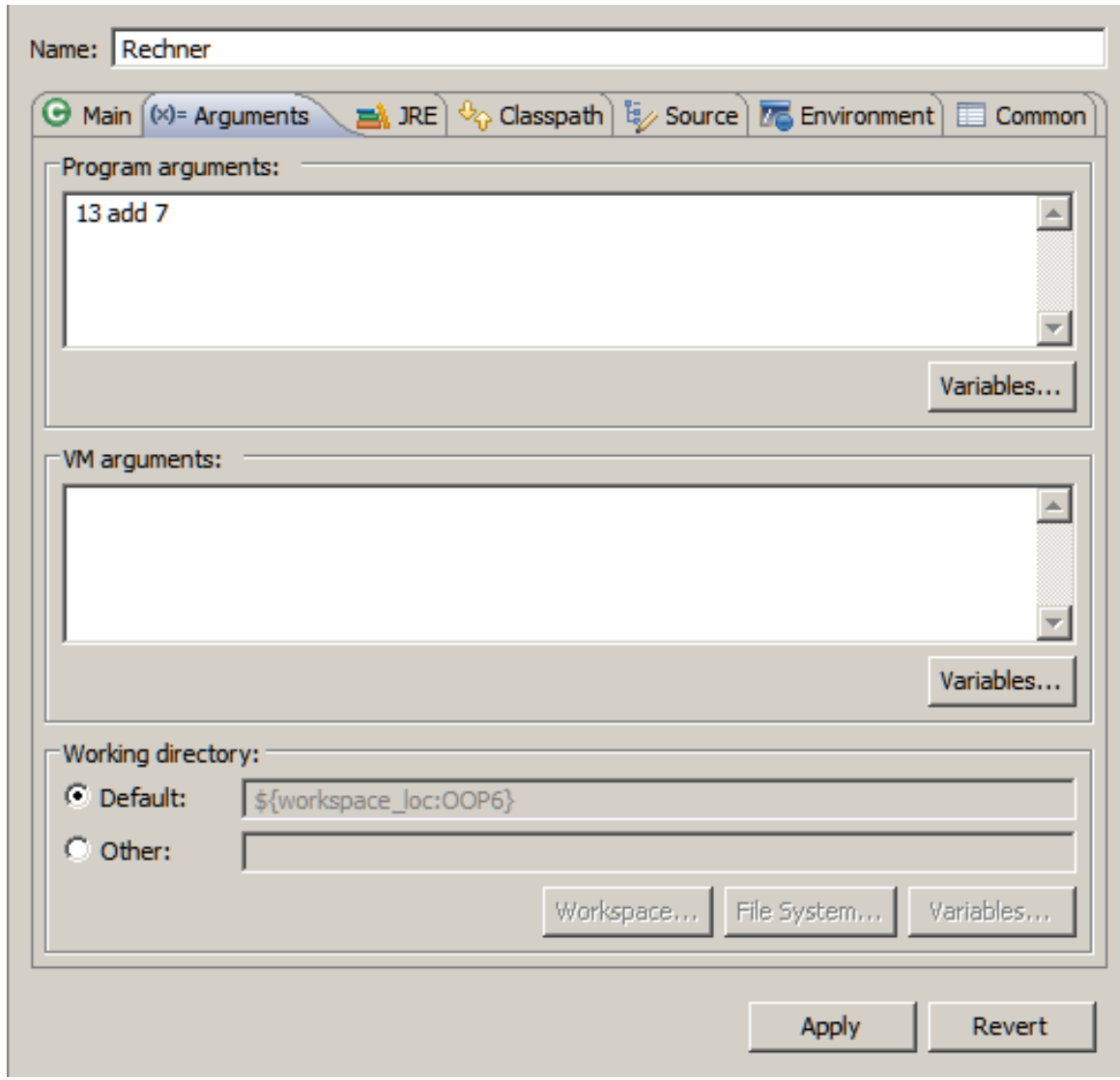
Fehlende Stellen sind durch ***HIER ERGÄNZEN*** markiert.

Aufgaben

Aufgabe 06.07 – Parameterübergabe bei Programmaufruf

In Eclipse:

Run → Run Configurations…



Aufgaben

Aufgabe 06.07 – Einfacher Taschenrechner mit Parameterübergabe

Entwickeln Sie einen einfachen Rechner, der die vier Grundrechenarten (+, -, *, /) beherrscht. Dieser Rechner soll durch Parameter beim Programmaufruf gesteuert werden. Hierzu werden dem Programm beim Aufruf zwei Zahlen als Strings sowie ein Schlüsselwort für die durchzuführende Operation übergeben. Diese Zahlen können mit Hilfe der Methode parseInt() der Wrapper-Klasse Integer in einen int-Wert gewandelt werden.

Die Reihenfolge der Parameter ist folgendermaßen definiert:

[Zahl1] [Operation] [Zahl2]

Verwenden Sie die folgenden Schlüsselworte für die Rechenoperationen:

add Addition: Zahl1 + Zahl2

sub Subtraktion: Zahl1 – Zahl2

mul Multiplikation: Zahl1 * Zahl2

div Division: Zahl1 / Zahl2

Ein Aufruf des Programms könnte beispielsweise so aussehen:

java Rechner 13 add 7

und würde zu folgendem Ergebnis führen:

13 add 7 ist 20