Programmiervorkurs 2006

1. Solaris
2. Programmkonstrukte
3. Grundlagen der Programmierung in Java
4. Javadoc
5. Einführung in Eclipse
6. Debugging

1. Solaris

Markus Durzinsky

Mardur AT gmx DOT net http://www-e.uni.magdeburg.de/durzinsk

0. Gliederung

- Solaris Betriebssystem
  - 1. Graphische Oberfläche
  - 2. Konsole
  - 3. Fernzugriff
  - 4. Links

1. Graphische Oberfläche – Untergliederung

- Solaris
  - 1. Graphische Oberfläche
    - 1.1 Graphische Anmeldung
    - 1.2 Dateien verwalten
    - 1.3 Im Internet surfen
    - 1.4 eMail lesen
    - 1.5 Dateien bearbeiten

1.1 Graphsche Anmeldung

- Modernes Mehrbenutzersystem
- Anmeldung mit Name und Passwort
- Hier im SUN-Pool
  - Thin-Clients nur als Arbeitsplatz
  - Rechenarbeit erledigt ein Server

1.2 Dateien verwalten

- Dateimanager zeigt Dateien und Verzeichnisse an
- drag&drop zum Verschieben
- Anzeigen von Dateien, direktes Starten des zugeordneten Programmes

1.3 Im Internet surfen

- Browser Firefox oder Mozilla
- Unter Optionen/Einstellungen den Festplatten-Cache deaktivieren
- Bookmarks anlegen
  - Java API
    http://java.sun.com/j2se/1.5.0/docs/api
  - Noch was? FIN-Seite, Univ-IS?

1.4 eMails lesen

- eMail-Adresse für Studenten
  alice@cs.uni-magdeburg.de
- Programm zum Schreiben von eMails
- Automatische Benachrichtigung bei neuen eMails
- eMails können automatisch an eine andere Adresse weitergeleitet werden (siehe Konsolenbefehle)

1.5 Dateien bearbeiten

- **`gedit`** oder Standardeditor des DE
- **`emacs`** Editor für Programmierer mit vielen Extras

- StarOffice kann auch Word-Dokumente öffnen

2. Konsole – Untergliederung

- Solaris
  - 2. Konsole
    - 2.1 Was ist das?
    - 2.2 Dateien verwalten
    - 2.3 Speicherbeschränkung mit Quota
    - 2.4 Zugriffsrechte
    - 2.5 Datenverarbeitung und Weiterleitung
    - 2.6 Suchen und Finden
    - 2.7 Prozesse verwalten
    - 2.8 Nützliche Programme

2.1 Konsole – Was ist das?

- Konsole ist die Textschnittstelle zu Solaris
- Man kann (fast) alles damit machen
- Man befindet sich in einem Verzeichnis
- Und führt einzelne Befehle aus oder startet Programme

2.1 Konsole - Hilfe!

- Informationen zu einem Programm erhält man durch Eingabe von
  - **`man`** ***`program`***
  - **`info`** ***`program`***
- Und wenn man den Programmnamen nicht kennt?
  - **`apropos`** ***`keyword`***
- Automatische Vervollständigung von Datei- und Programmnamen mit der Tabulator-Taste

2.2 Dateien verwalten – Navigation

- Eigenes Nutzerverzeichnis
  - **`/home/alice`** (Kurzform **`~/`**)
- **`pwd`** aktuelles Verzeichnis ausgeben
- **`ls`** Inhalt ausgeben
- **`cd`** ***`dir`*** Verzeichnis wechseln
- Spezielle Verweise
  - **`.`** aktuelles Verzeichnis
  - **`..`** übergeordnetes Verzeichnis
- **`cd`** zurück zum Nutzerverzeichnis

2.2 Dateien verwalten – Erstellen und Löschen

- **`touch file`** erstellt eine neue Datei
- **`rm file`** löscht die Datei
  - Warnung: keine Nachfrage und keine Wiederherstellung möglich
- **`mkdir dir`** erstellt ein neues Verzeichnis
- **`rmdir dir`** löscht ein leeres Verzeichnis
  - **`rm -r dir`** löscht auch ganze Verzeichnisse mit Inhalt

2.3 Speicherbeschränkung mit Quota

- Speicherplatz für Dateien ist begrenzt
  - Softlimit darf nur für einen kurzen Zeitraum überschritten werden
  - Hardlimit darf (und kann) nicht überschritten werden
- **`quota -v`** zeigt die aktuelle Belegung und die Schranken an
- **`du -h`** zeigt alle Verzeichnisse und deren Größe an

2.4 Zugriffsrechte – Anzeigen und Verstehen

- Dateien gehören einem Besitzer und einer Gruppe
- Mögliche Ausgabe von **`ls -l`**

```
drwxr-xr-x alice stud   24 2006-09-10 15:09 verzeichnis
-rw-r--r-- alice stud 1035 2006-09-12 21:25 datei.txt
```

  - **`d`** Verzeichnis
  - **`rwx`** Zugriffsrechte Besitzer, Gruppe, Andere (Lesen, Schreiben, Ausführen)
  - Name des Besitzers und der Gruppe
  - Größe, Änderungsdatum und Dateiname

2.4 Zugriffsrechte – Ändern

- **`chown user file`** ändert den Besitzer
- **`chmod changes file`** ändert die Zugriffsrechte
- Änderungen symbolisch
  - **`go+w`** gibt Gruppe und Anderen Schreibrechte
- Oder numerisch
  - **`640`** gibt dem Besitzer Schreibrechte, Gruppe darf lesen, Andere nicht

2.4 Zugriffsrechte – Ändern

- Symbolische Rechte
  - **u** Besitzer, **g** Gruppe, **o** Andere, **a** Alle
  - **+** Rechte geben, **-** Rechte entziehen
  - **r** Lesen, **w** Schreiben, **x** Ausführen
- Numerische Rechte
  - Drei Ziffern jeweils bestehend aus
    4 + 2 + 1
    Lesen + Schreiben + Ausführen

2.5 Datenverarbeitung – Textausgabe

- **cat *file file ...*** gibt den Inhalt aller Dateien hintereinander aus
- **more *file*** gibt die Datei aus, wartet nach jeder vollen Seite
- **less *file*** wie **more**, Zurückblättern möglich
- **echo *text*** gibt den Text wieder aus

2.5 Datenverarbeitung – Textausgabe

- **head *file*** gibt die ersten Zeilen aus
- **tail *file*** gibt die letzten Zeilen aus
  - Mit der Option **-f** wird die Datei überwacht und Änderungen automatisch weiter ausgegeben
  - **-n anzahl** Nur so viele Zeilen ausgeben
- **sort *file*** sortiert die Zeilen lexikographisch
  - Mit der Option **-n** wird numerisch sortiert

2.5 Datenverarbeitung – Ausgabe umleiten

- Mit **>** wird die Ausgabe in eine Datei umgeleitet werden
  - **echo "alice@gmx.net" > .forward**
    Schreibt die Adresse in die Datei
    (Dadurch werden alle eingehenden eMails an diese Adresse weitergesendet)
  - **sort datei.txt > datei2.txt**
    Schreibt die Zeilen der Textdatei sortiert in eine neue Datei
- Mit **>>** wird die Ausgabe an die Datei angehangen

2.5 Datenverarbeitung – Ausgabe weiterleiten

- Mit **`|`** wird die Ausgabe des ersten Befehls als Eingabe des zweiten verwendet
- **`>`**, **`>>`**, **`|`** lassen sich auch kombinieren
  - **`du | sort -n`**
    Verzeichnisse sortiert nach Größe
  - **`du | sort -n | tail > datei.txt`**
    speichert die Namen der 10 größten Verzeichnisse

2.6 Suchen und Finden

- **`find`** sucht nach Dateien mit bestimmten Eigenschaften
  - **`find`** alle Dateien im aktuellen Verzeichnis, inklusive Unterverzeichnisse
  - **`find -name *.java`** Dateien mit diesem Muster im Dateinamen
- **`grep`** sucht Inhalte in Dateien
  - **`grep test *.java`** Java-Dateien, welche das Wort 'test' enthalten

2.7 Prozesse verwalten

- **`STRG-c`** bricht den aktuellen Befehl ab
- **`STRG-z`** legt den aktuellen Befehl schlafen
  - **`fg`** führt den letzten schlafenden Prozess weiter aus
  - **`bg`** wie **`fg`**, jedoch ohne die Konsole zu belegen
  - **`jobs`** zeigt alle schlafenden Prozesse an

2.7 Prozesse verwalten

- **`kill *pid*`** beendet einen Prozess
  - Dies ist das letzte Mittel, falls nichts anderes mehr funktioniert
  - Die Prozessnummer liefern diese Befehle
- **`ps -af`** zeigt alle in einer Konsole laufenden Prozesse
- **`top`** Zeigt eine aktualisierende Tabelle aller Prozesse an (auch CPU-Last)
  - Taste **`k`** beendet Prozesse

2.8 Nützliche Programme

- **`file filename`** zeigt den Dateityp an, selbst bei falscher Endung
- **`w`** zeigt alle am System angemeldeten Benutzer
- **`finger user`** Informationen über einen Benutzer, wie der vollständige Name

2.8 Noch mehr nützliche Programme

- **`nispasswd`** ändert das Anmelde-Passwort
- **`alias`** listet alle Synonyme auf
- **`alias l="ls -al"`** erstellt ein neues Synonym

2.8 Nützliche Programme – Kompression

- **`zip archiv text.txt ...`** komprimiert nach **`archiv.zip`**
- **`zip -r archiv dir`** komprimiert alle Dateien im Verzeichnis
- **`unzip -l archiv`** listet alles Dateinamen auf
- **`unzip archiv`** entpackt alle Dateien in das aktuelle Verzeichnis

3. Fernzugriff – Untergliederung

- Solaris
  - 3. Fernzugriff
    - 3.1 Terminalanmeldung
    - 3.2 Graphische Programme
    - 3.3 Dateien übertragen

3.1 Terminalanmeldung

- **`ssh` *`computer-name`***
  - Verbindung mit einem anderen Rechner über das Netzwerk herstellen
  - Passwort wird abgefragt
  - Man erhält ein Terminal, welches wie eine lokale Konsole funktioniert
- Unter Windows erledigt Putty die Aufgabe der lokalen Konsole

3.2 Graphische Programme

- **`ssh -X` *`alice@computer`***
  - Nun können sogar graphische Programme gestartet werden
  - Diese werden auf dem lokalen Rechner angezeigt
  - Aber auf dem entfernten Rechner ausgeführt
- Funktioniert auch mit Putty, wenn man einen XServer hat (etwa cygwin oder xming)

3.3 Dateien übertragen

- **`sftp` *`alice@computer`***
  - Hiermit können Dateien zwischen zwei Rechnern übertragen werden
- **`gftp`** ist die graphische Variante
- Unter Windows kann man WinSCP verwenden

4. Links

- Windows-Programme zum Zugriff auf Solaris-Rechner
  - Putty – kostenloses sshhttp://www.chiark.greenend.org.uk/~sgtatham/putty/
  - Xming – freier XServer (Graphikoberfläche für Solaris-Programme) http://www.straightrunning.com/Xming/
  - WinSCP – kostenloses sftp http://www.winscp.net

2 Programmkonstrukte

Gerhard Gossen

Inhalt

2.0 Programmkonstrukte

- Programme bestehen aus Kombinationen folgender Grundkonstrukte:
  - Sequenz
  - Bedingte Ausführung / Fallunterscheidung
  - Wiederholung
- Daraus lässt sich *jedes* Programm bilden

2.0 Notationen

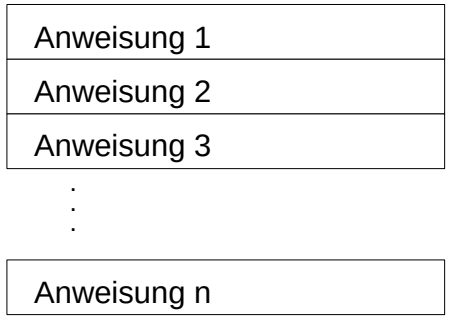
- Verschiedene Notationen (Schreibformen)
  - Umgangssprachlich
  - Struktogramm
  - Pseudo-Code
  - Code in einer Programmiersprache
- Verwendete Konventionen
  - **Fett**: feste Zeichenketten, Syntax-Bausteine
  - *Kursiv*: Variablennamen u.ä.

2.1 Sequenz

- Nacheinander Ausführen mehrerer Anweisungen
- Bsp.

```
Führe Anweisung 1 aus, danach
  Anweisung 2, [...], führe
  letzte Anweisung aus.
```

2.1 Sequenz



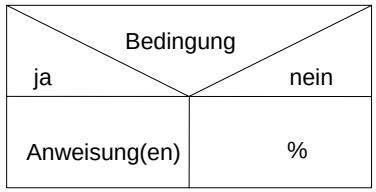
2.1 Sequenz

```
Anweisung 1;
 Anweisung 2;
 Anweisung 3;
        .
        .
 Anweisung n;
```

2.2.1 Fallunterscheidung: einseitig

- Ausführung abhängig von einer Bedingung
- Bsp.
  `Nur wenn ... gilt, führe Anweisung aus`

2.2.1 Fallunterscheidung: einseitig

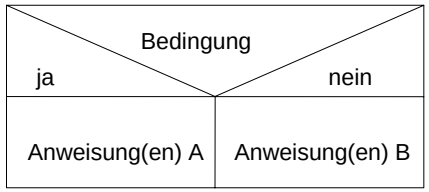


2.2.1 Fallunterscheidung: einseitig

```
IF Bedingung
  THEN
    Anweisungen
  ENDIF
```

2.2.2 Fallunterscheidung: zweiseitig

- Variante:
  Wenn Bedingung gilt, dann A;
  ansonsten B.



2.2.2 Fallunterscheidung: zweiseitig

```
IF Bedingung THEN
    Anweisungen
ELSE
    Anweisungen
ENDIF
```

2.2.3 Fallunterscheidung: mehrseitig

- Variante 2: Mehrere Fälle
  Wenn Selektor den Wert a hat, Anweisung A, bei Wert b Anweisung B, ... , bei Wert n Anweisung N.

Falls s =				
1	2	3	...	n
a_1	a_2		a_3 ...	a_n

- Selektor ist z. B. eine Variable

2.2.3 Fallunterscheidung: mehrseitig

```
IF Selektor =
    1: Anweisung 1
    2: Anweisung 2
    ...
    n: Anweisung n
ENDIF
```

2.3 Wiederholung

- Wiederholen von Anweisungen
  - Abhängig von einer Bedingung
  - *n*-mal
- `Solange Bedingung A gilt, wiederhole Anweisung(en)`

2.3.1 Wiederholungsschleife

- Eingangsbedingung:
  Überprüfung *vor Beginn* der Schleife

Solange A	
tue	Anweisung(en)

2.3.1 Wiederholungsschleife

- Ausgangsbedingung:
  Überprüfung *nach Ende* der Schleife

Wiederhole	Anweisung(en)
bis A	

- Wird mindestens einmal ausgeführt

2.3.1 Schleife mit Eingangsbedingung

```
WHILE Bedingung DO
    Anweisung(en)
ENDWHILE
```

2.3.1 Schleife mit Ausgangsbedingung

```
REPEAT
    Anweisung(en)
UNTIL Bedingung
```

2.3.2 Zählschleife

- Kurzschreibweise einer `while`-Schleife
- Häufiger Anwendungsfall: Für jeden Wert zwischen *a* und *b*

Für i = anfw bis endw	
tue	a

2.3.2 Zählschleife (for-Schleife)

```
FOR i := Anfangswert TO Endwert
  DO
    Anweisungen
```

2.4 Kombination

- Die Konstrukte lassen sich beliebig tief ineinander verschachteln
- Beispiel:

```
FOR i :=  1 TO 10 DO
  IF i gerade
    Gib "i ist gerade" aus
  ELSE
    Gib "i ist ungerade" aus
  ENDIF
ENDFOR
```

3. Grundlagen der Programmierung in Java

1. Grundgerüst eines Javaprogramms
2. Variablen und Datentypen
3. Strings
4. Arrays
5. Kontrollstrukturen
6. Methoden
7. Klassen und Objekte

3.1 Grundgerüst – Programmaufbau

- Programmbeispiel *HelloWorld.java*

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello World!");
      // Dieses Programm gibt „Hello World!“
aus
   }
}
```

3.1 Grundgerüst – Programmaufbau

- **`public class HelloWorld {`**
  - Jedes Programm besteht aus mindestens einer öffentlichen Klasse (hier: *HelloWorld*)
  - Klassenname = Dateiname
    - public class HelloWorld ⇔ HelloWorld.java

3.1 Grundgerüst – Programmaufbau

- **`public static void main(String[] args) {`**
  - Einsprungpunkt des Java-Interpreters
    - Main-Methode der aufgerufenen Klasse wird automatisch ausgeführt
  - Jede Klasse hat genau eine main-Methode
  - **`Args`** ist Array aus Kommandozeilenparametern

3.1 Grundgerüst – Programmaufbau

- `System.out.println("Hello World!");`
  - Anweisungen in Java werden mit `;` abgeschlossen
  - Diese gibt „Hello World!“ auf der Konsole aus
- `// Dieses Programm gibt "Hello World!" aus`
  - Einzeilige Kommentare werden mit `//` eingeleitet
  - Mehrzeilige Kommentare

```
/* Dieses Programm
gibt "Hello World!" aus */
```

3.1 Grundgerüst – Kompilieren und Ausführen

- Konsolenbefehl
  - Kompilieren

    `Prompt :> javac HelloWorld.java`

    Allgemein: `Prompt :> javac [Quelltext.java]`
  - Ausführen

    `Prompt :> java HelloWorld`

    Allgemein: `Prompt :> java [Klassenname]`

    Aufgerufene Klasse muss main-Methode beinhalten!

3.2 Variablen und Datentypen – Variablen

- Variable = Name für Speicherbereich
- Variablen müssen deklariert werden!
  - Syntax: `typ variablenname;`
  - Z.B. `int x;`
- Verwendung ohne vorherige Deklaration nicht möglich!

```
// int x;
x = 2; // Kompilierfehler!
```

3.2 Variablen und Datentypen – Datentypen

Typ	Größe (bit)	Bereich
byte	8	-128 bis 127
short	16	-32.768 bis 32.767
int	32	-2.147.483.648 bis 2.147.483.647
long	64	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
char	16	`'\u0000'` bis `'\uFFFF'`
float	32	$\pm 3{,}40282347 \times 10^{38}$
double	64	$\pm 1{,}7976931348623157 \times 10^{308}$
boolean	1	{`true`; `false`}

3.2 Variablen und Datentypen – Über- und Unterlauf

- Über- bzw. Unterschreitung des Wertebereichs eines Datentyps
- Beispiel

```
short x = 32767; // Größtmöglicher Wert
x++;
System.out.println(x); // Ausgabe:-32768
```

3.2 Variablen und Datentypen – Rundungsfehler

- Rundungsfehler durch begrenzte Größe der Binärdarstellung einer Gleitkommazahl
- Beispiel

```
double cosinus =
Math.cos(Math.toRadians(90.0));
if (cosinus == 0.0)
   System.out.println(„Gleich 0.0!“);
else {
    System.out.println(„Ungleich 0.0!“);
    System.out.println(cosinus);
}
```

3.3 Strings

- Objekte der Klasse String
- Lassen sich wie elementare Datentypen behandeln

3.3 Strings

- Erzeugung:
  - Durch Instanzenbildung

```
String str = new String(); // leerer
String ""
String str = new String("Hallo");
```

  - Durch Literale

```
String str = "Hallo";
```

    - Keine Mehrzeiler! ⇒ Konkatenation mit +

```
String str = "Das ist ein " +
                    Mehrzeiler";
```

3.3 Strings

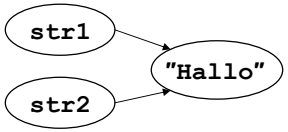
- Operationen

```
String a = "Das ist " + "ein Test";
String b = a + " mit Strings";
```

- Zuweisungen

```
String str1 = new String("Hallo");
String str2;
str2 = str1;
```

Verweis auf das selbe String Objekt



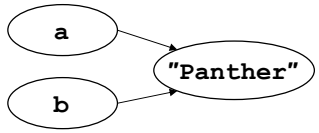
3.3 Strings

- Vergleiche
  - `==` und `!=`
    - Bei Erzeugung durch Literale

```
String a = "Panther";
String b = "Panther";
```

`a == b` liefert `true`

`a != b` liefert `false`



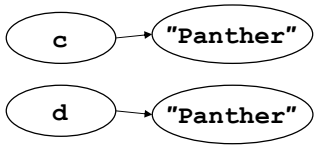
3.3 Strings

- Vergleiche
  - `==` und `!=`
    - Bei Erzeugung durch Instanzen

```
String c = new String("Panther");
String d = new String("Panther");
```

`c == d` liefert `false`

Verschiedene Referenzen!



3.3 Strings

- Vergleiche
  - `compareTo()` und `equals()`
    - String-Methoden:

`c.compareTo(d) == 0` liefert `true`

`c.equals(d)` liefert `true`

- Funktionieren immer ⇒ Mit den Stringeigenen Vergleichmethoden `compareTo()` und `equals()` ist man auf der sicheren Seite!

3.4 Arrays

- Erzeugung
  - Syntax:
    `typ[] name = new typ[Anzahl_Elemente];`
  - Bsp.: `double[] feld = new double[12];`
    Feld mit 12 double-Werten
  - Alternativ Initialisierungsliste
    `double[] feld = {0.1, 2.3, 1.7, 5.9};`
  - Größe des Feldes steht bei Initialisierung fest!

3.4 Arrays

- Zugriff

```
int[] zahlen = new int[6];
```

0	1	2	3	4	5

Index ∈ [0; Anzahl_Elemente-1]

```
zahlen[0] = 13;
zahlen[1] = 4;
...
zahlen[5] = 43;
```

- Array-Größe

```
int a = zahlen.length;    // a = 6
```

3.5 Kontrollstrukturen – Bedingungen

- `if (Bedingung) {`
  `Anweisungen;`
  `}`
- Bedingung ist logischer Ausdruck
  ⇒ Ergebnis ist **`true`** oder **`false`**
- Bsp.: Wert an sich (**`true`**, **`false`**); Vergleiche (**`x>100`**); Methoden, die **`true`** oder **`false`** zurückgeben (**`c.equals(d)`**)

3.5 Kontrollstrukturen – Bedingungen

- Logische Operatoren
  - A, B seien logische Ausdrücke und ⊗ ein logischer Operator => A ⊗ B ist logischer Ausdruck
  - UND - &&
    - A && B ergibt **`true`**, falls A und B jeweils **`true`** ergeben

3.5 Kontrollstrukturen – Bedingungen

- Logische Operatoren
  - OR - ||
    - A || B ergibt **`true`**, falls A oder B oder beide **`true`** ergeben
  - NOT - !
    - !A ergibt **`true`**, falls A **`false`** ergibt

3.5 Kontrollstrukturen – Verzweigung

- **`if (Bedingung) {`**
  `Anweisungen;`
  `}`
- Bsp.:

```
if (x == 3) {
   System.out.println("x ist 3");
}
```

3.5 Kontrollstrukturen – Verzweigung

- **`if (Bedingung) {`**
  `Anweisungen;`
  `}`
  `else {`
  `Anweisungen;`
  `}`
- Bsp.:

```
if (x == 3) {
   System.out.println(„x ist 3“);
}
else {
   System.out.println(„x ist nicht 3“);
}
```

3.5 Kontrollstrukturen – switch-Verzweigung

- **`switch (Ausdruck) {`**
  `case KONST1:` **`Anweisungen;`**
  `break;`
  `case KONST2:` **`Anweisungen;`**
  `break;`
  `...`
  `default:` **`Anweisungen;`**
  `break;`
  `}`
- Ausdruck ist numerisch, nicht logisch

3.5 Kontrollstrukturen – switch-Verzweigung

- Bsp.:

```
int x;
switch (x) {
case 1:   System.out.println("x ist 1");
          break;
case 2:   System.out.println("x ist 2");
          break;
default:  System.out.println("x ist " +
                    "was anderes");
          break;
}
```

3.5 Kontrollstrukturen – while-Schleifen

- `while (Bedingung) {`
  `Anweisungen;`
  `}`
- Bsp.:

```
int n = 0;
while (n <= 10) {
   System.out.println("2 hoch " + n +
                " ist " + Math.pow(2,n));
    n++;
}
```

3.5 Kontrollstrukturen – do-while-Schleifen

- `do {`
  `Anweisungen;`
  `} while (Bedingung);`
- Bsp.:

```
int n = 0;
do {
   System.out.println("2 hoch " + n +
                " ist " + Math.pow(2,n));
    n++;
} while (n <= 10);
```

3.5 Kontrollstrukturen – for-Schleifen

- `for (Initialisierung; Bedingung; Veränderung) {`
  `Anweisungen;`
  `}`
- Bsp.:

```
for (int n = 0; n <= 10; n++) {
   System.out.println("2 hoch " + n +
                " ist " + Math.pow(2,n));
}
```

3.5 Kontrollstrukturen – for-Schleifen

- Jede for-Schleife lässt sich auch als while-Schleife ausdrücken:

```
for (Initialisierung; Bedingung;
Veränderung) {
   Anweisungen;
}
```

lässt sich ausdrücken als

```
Initialisierung;
while (Bedingung) {
   Veränderung;
}
```

3.5 Kontrollstrukturen – Endlosschleifen

- Achtung: Die Abbruchbedingung muss immer erreicht werden!
- Bsp.:

```
int n = 0;
while (n <= 10) {
   ...
   n--;
}
```

3.6 Methoden

- Abtrennung von Aufgaben in Funktionen (in Java „Methoden“ genannt)
- Analogie zur Mathematik:
  - y = f(x)
    - x – Parameter
    - y – Rückgabewert,
    - f – Methode
  - $f(x) = x^2$

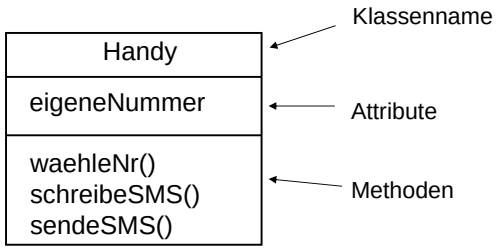
```
double f (double x) {
   return x*x;
}
```

3.7 Klassen und Objekte

- In Java allgegenwärtig (Klasse *HelloWorld*, Strings, Math, …)
- Kapselung von Daten und Methoden
  - Nutzer der Klasse muss nicht wissen, wie das Innere aussieht (z.B. Strings: `str1.equals(str2)`)
- Erleichtert Wiederverwendung und Weitergabe an andere Programmierer

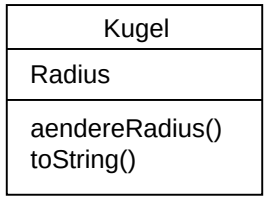
3.7 Klassen und Objekte

- Klassen bestehen aus Attributen und Methoden
- Bsp.: *Handy*



3.7 Klassen und Objekte

- Jetzt ein praktisches Beispiel: *Kugel*



3.7 Klassen und Objekte

```
public class Kugel {
   private float radius;
   public Kugel() {
      radius = 0.0;
   }
   public Kugel(float r) {
      radius = r;
   }
   ...
```

3.7 Klassen und Objekte

```
   ...
   public aendereRadius(float r) {
      radius = r;
   }
   public String toString() {
      return "Kugel: Radius = " + radius;
   }
}
```

3.7 Klassen und Objekte

- Nutzung der Klasse *Kugel* im Programm

```
public class kugelTest {
    public static void main(String[] args) {
        Kugel k1 = new Kugel();
        Kugel k2 = new Kugel(10.0);
        k1.aendereRadius(3.5);

        System.out.println(k1);
        // Ausgabe "Kugel: Radius = 10.0"
        System.out.println(k2);
        // Ausgabe "Kugel: Radius = 3.5"
    }
}
```

4. Javadoc

Erstellung und Nutzung von Java API-Dokumentationen

Inhalt

- Was ist Javadoc?
- Was ist eine API-Dokumentation?
- Wozu eine API-Dokumentation?
- Die Java-API
- Erzeugung von Java API-Dokumentationen
- Java-Namenskonventionen

Was ist Javadoc?

- entwickelt von *Sun Microsystems*
- Programm zur Erstellung von Java API-Dokumentationen
- parst die in einem Quelltext enthaltenen Kommentare
- erzeugt daraus Struktur von HTML-Dateien, die Java API-Dokumentation

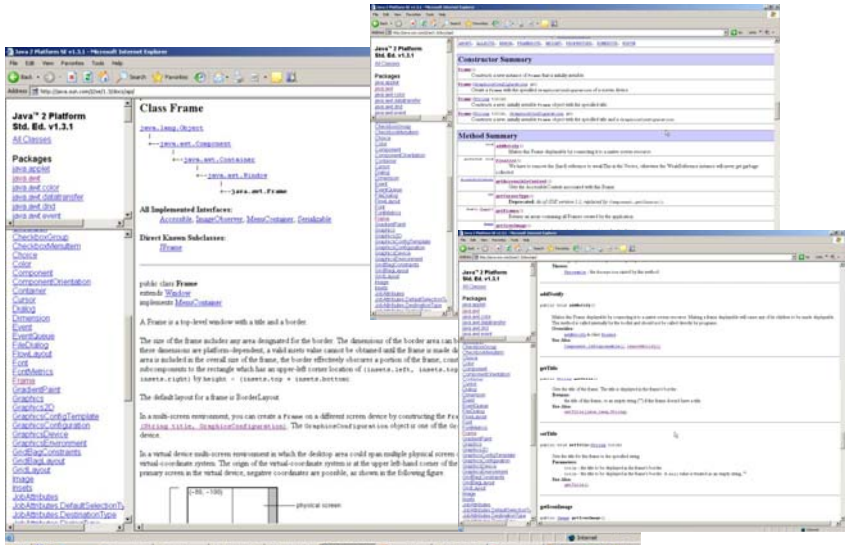
Was ist eine API-Dokumentation?

- enthält Erklärung der von einer Programmierschnittstelle zur Verfügung gestellten Methoden und Klassen
- beschreibt deren Funktionsweise, Rückgabewert, Art und Anzahl von Parametern
- Darstellung einer Klassen- hierarchie und aller vorhandenen Methoden
- Beispiel:
  - http://java.sun.com/j2se/1.5.0/docs/api/

Wozu eine API-Dokumentation?

- Klassen und Methoden werden meist von mehreren Programmierern genutzt
- Nachvollziehen der Funktionsweise im Quelltext zu zeitintensiv
- schnelleres Finden von Methoden die etwas Bestimmtes tun

Die Java-API



Erzeugung API-Dokumentationen

- Beschreibende Kommentare VOR allen öffentlichen Klassen, Schnittstellen, Methoden und Variablen
- Dokumentationskommentare beginnen mit „/**“
- Erster Satz wird in *Method Summary* dargestellt
- Alles andere steht später in *Method Detail*
- Erweiterungen um Verweise, Versionen etc.

Erzeugung API-Dokumentationen

- Beispiel:

```
/**
  * Die ist der kurze Satz im Method Summary
  *
  * Dieser Satz steht dann bereits im Method Detail,
  * zusammen mit allen folgenden Sätzen und
  * Erweiterungen
  * @version 0.1b
  */

public void dofoo() {}
```

Erzeugung API-Dokumentationen

- Erweiterungen:
  - ***@return Rückgabetext***

    Beschreibung des Rückgabewert einer Methode
  - ***@see (Klasse/Methode)***

    Querverweis auf eine andere Klasse oder Methode
  - ***@version Versionstext***

    Angabe der Version einer Klasser oder Methode

Erzeugung API-Dokumentationen

- Erweiterungen:
  - ***@param NameParamter Text***

    Beschreibung der Parameter einer Methode
  - ***@exception Klasse Text***

    Beschreibung der von einer Klasse geworfenen Exceptions
  - ***@author Text***

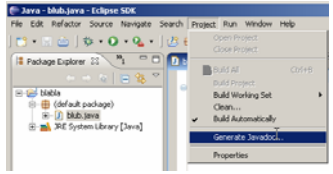
    Author einer Methode oder Klasse

Erzeugung API-Dokumentationen

- Sonderfall „*deprecated*“
  - Angebotene Methoden werden durch neue ersetzt
  - Namen von Funktionen wurden bei der Erstellung falsch geschrieben
  - Nachträgliche Anpassung an Java-Namens-konventionen
  - ***Problem: Methoden können nicht einfach gelöscht werden***
  - ***Lösung: Kennzeichnung mit Erweiterung (@deprecated)***

Erzeugung API-Dokumentationen

- Aufruf erfolgt von der Kommandozeile aus:
  - *javadoc AdieuDuSchnoedeWelt.java* (oder *.java für alle Quelltextdateien im aktuellen Verzeichnis)
- Javadoc in vielen Entwicklungsumgebungen (Netbeans, Eclipse) integriert



- Erzeugung anderer Dateiformate mittels *Doclets*

Java-Namenskonventionen

- sollen Einarbeitung in fremde Quelltexte erleichtern
- Java unterscheidet zwischen Groß- und Kleinschreibung
  - *z.B.:Variable1 ≠ variable1*
- Verwendung sogenannter *„sprechender Bezeichner“*
  - *z.B.*: *ArrayIndexOutOfBoundsException*
- Konstanten vollständig groß geschrieben
  - z.B.: *Color.WHITE*

Java-Namenskonventionen

- Bezeichner von Klassen mit großen Anfangsbuchstaben
- bei aus mehreren Wörtern bestehend Bezeichnern, jeder Anfangsbuchstabe groß
  - z.B.: *LayoutManager*
- Bezeichner von Variablen und Methoden mit kleinen Anfangsbustaben
- Anfangsbuchstaben folgender Wörter groß
  - z.B.: *Component.getPreferredSize()*

5. Einführung in Eclipse

Ivonne Schröter

Inhaltsverzeichnis

5.1 Installation

- **Quelle:**
  - http://www.eclipse.org
- **PlugIn-Installation:**
  - ..\eclipse\plugins
  - ..\eclipse\features
- **Wahl des Workspace** (Ordner für Projektspeicherung)

5.2 Umgebungsstruktur

- **verschiedene Umgebungen:**
  - anwählbar oben rechts in einer extra Toolbar
  - Java-Umgebung
  - Debug-Umgebung
  - CVS-Ungebung

5.3 neues Projekt

- **Projekte in Eclipse:**
  - Programm auch ohne Projekt funktionstüchtig
  - Eclipse braucht Zusatzinformationen um seine Funktionen entfalten zu können (Abspeicherung in Projekten)

5.3 neues Projekt

- **neues Projekt anlegen:**
  - Kontextmenü vom „Package Explorer“ aufrufen
  - NEW → PROJEKT
  - Java-Projekt anlegen
  - Name des neuen Projekts eintragen
  - FINISH anklicken

5.3 neues Projekt

- **Packages:**
  - Überblick behalten durch Anlegen von Packages (sein Projekt → NEW → Package)
  - zueinander gehörende Klassen in ein gemeinsames Package schieben
  - Jeder muss sein individuelles System entwickeln

5.4 Hello World

- **neue Klasse:**
  - „Projekt Explorer“ → auf sein neues Projekt klicken → NEW → CLASS
  - Name der Klasse eintragen (sollte mit einem Großbuchstaben beginnen)
  - Hacken bei:
    public static void main(String[ ] args)
  - Eclipse legt eine neue „.java–Datei“ an (Quellcode wird gespeichert)

5.4 Hello World

- **Main-Methode:**
  - Kommentare:
    - /* .. */
    - //
  - Java-Doc Kommentare:
    - /** .. */
  - HelloWorld-Programm:
    - in Methodenkörper „{ }“ einfügen
      - System.out.println(“Hello World“);

5.4 Hello World

- **Programm ausführen:**
  - Klick auf den kleinen Abspiel-Pfeil in der Symbolleiste (beim ersten Start: neben dem Programm-Start-Button → „Run...“ → Java Application auswählen → NEW)
  - Ergebnis: „Hello World“-Schriftzug im unten stehenden Kosolenfenster (hier erscheinen auch Fehlermeldungen)

5.5 Optionen

- **kompilieren, ohne auszuführen**
  - Menue „Projekte“ → Haken bei „Build Automatically“ weg machen
  - neuer Button erscheint
  - nicht mehr nötig das Programm auszuführen, um es zu kompilieren

5.5 Optionen

- **Textvervollständigung**
  - Window → Preferences... → Java → Editor → Templates
  - Regeln definieren:
    - Name: Abkürzung (z.B. sopl)
    - Pattern: Ersetzung (z.B. System.out.println(“”);)
    - Insert Variable: Wahl einer zusätzlichen Variablen (z.B. Cursor zwischen die Anführungsstriche)

5.6 Kleiner Tipp für EAD- Übungen

- **So hab ich es gemacht!**
  - **ein Projekt angelegt**
  - **für jede Aufgabe ein Package erzeugt**
  - **Falls notwendig, Packages einer anderen Aufgabe mit einbinden**

6. Debugging

von Julia Preusse

6.1 Programmfehler in der Geschichte

- **1962** führte ein fehlender Bindestrich in einem Fortran-Programm zum Verlust der Venus-Sonde Mariner 1, welche über 80 Millionen US-Dollar gekostet hatte.
- **1985 -1987** gab es mehrere Unfälle mit dem medizinischen Bestrahlungsgerät Therac-25. Infolge einer Überdosis, die durch fehlerhafte Programmierung und fehlende Sicherungsmaßnahmen verursacht wurde, mussten Organe entfernt werden, drei Patienten verstarben.
- **1999** verpasste die NASA-Sonde Climate Orbiter den Landeanflug auf den Mars, weil die Programmierer das falsche Maßsystem verwendeten – Yard statt Meter. Die NASA verlor dadurch die mehrere hundert Millionen Dollar teure Sonde.
- **2002** schalteten sich Siemens Mobiltelefone im April bei Aufruf der Kalenderfunktion ab. Bekannt ist der Fehler auch als Aprilbug. Als Folge könnte angesehen werden, dass der Mobiltelefonhersteller von dem Nutzen eines benutzergesteuerten Software Update überzeugt wurde.

6.2 Fehlerarten

- Syntaktische Fehler
- Laufzeitfehler
- Logische Fehler
- Versteckte Fehler

6.2.1 Syntaktische Fehler

- **verhindern bei Java bereits Kompilieren**
- Beispiele:
  - Falsch geschriebene Objekte und Methoden (WICHTIG Beachtung von Groß-und Kleinschreibung)
  - Falsch gesetzte Klammern (Begin/End, Text...)
  - Fehlerhafte Semikolons
  - Falsch importierte Packages
  - ...

6.2.1 Syntaktische Fehler

- werden bei Eclipse sofort am Rand mit folgenden Symbolen markiert:

Syntaktische Fehler sind trivial: sie sind werden sofort gefunden und können leicht beseitigt werden.

6.2.2 Laufzeitfehler

- **Fehler die erst beim Ausführen des Programms auftreten**
- Beispiele:
  - fehlende aber benötigte Parameter (z.B Eingabedatei)
  - Lesezugriff auf nicht initialisierte Variable (bzw. auf Objekte)
  - Zugriffe auf nicht vorhandene Speicherstellen
  - illegale Typumwandlung
  - Referenzierung durch Nullpointer
  - ungültige arithmetische Operation
  - nicht erreichbarer Code

6.2.2 Laufzeitfehler

- bei Laufzeitfehlern werden so genannte Exceptions geschmissen

  Die Häufigsten hiervon sind:
  - NullPointerException
    (wenn man versucht auf einen Verweis zuzugreifen, der den Wert null hat)
  - ArrayOutOfBoundException
    (bei ungültigen Feldzugriff)
  - BufferOverflowException
    (wenn das Pufferlimit erreicht; Endlosrekursion)
  - ArithmeticExeption
    (Division durch 0, Wurzel negative Zahl...)

- das Programm terminiert nicht; es muss mit dem terminate-Button beendet werden

6.2.2 Laufzeitfehler

Wichtig ist es, die Fehlermeldungen ordentlich zu lesen; bei Eclipse wird zusätzlich noch die Zeile, in der die Ausnahme aufgetreten ist, aufgeführt.

6.2.3 Logische Fehler

- WICHTIG: Herausfinden, wann der Fehler bzw. für welche Testdaten der Fehler auftritt.
- erst einmal Programm durchlesen bzw. überfliegen und nachsehen ob
  - ein Zahlendreher
  - eine falsche Formel
  - oder ein einfacher Denkfehler

  vorliegt.

Wenn das Programm danach immer noch nicht ordnungsgemäß läuft, geht das eigentliche Debugging los.

6.2.3 Logische Fehler

Jede Programmiersprache hat ihre kleinen Eigenheiten, deswegen gut in der Vorlesung aufpassen ;)

6.2.3 Logische Fehler

Aufgabenstellung:

Implementieren Sie eine Methode *swap(int a, int b)*, die die beiden Werte a und b miteinander vertauscht.

6.2.3 Logische Fehler

Idee trivial:

```
public void swap (int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
```


6.2.3 Logische Fehler

Aber beim Testen der Funktion:

Die Werte werden außerhalb der *swap*-Methode nicht mit einander vertauscht.

☹

6.2.3 Logische Fehler

Swap-Methode vertauscht nur die Werte miteinander, speichert diese aber nicht bei den neuen Speicherstellen ab.

Wenn man dementsprechend außerhalb der Swap-Methode auf die Originalwerte zugreift, dann erhält man die unvertauschten Werte.

-> „Pass by value“ (Übergabe des Wertes)

6.2.3 Logische Fehler

- Übergabe des Wertes: der Parameterliste wird eine Kopie der Werte der Argumente übergeben.
- Übergabe durch Referenz : die Speicheradresse des Wertes des Argumentes wird übergeben. Wird der Wert durch die aufgerufene Routine geändert, bleibt die Änderung des Wertes auch nach Verlassen der Routine bestehen.

Java unterstützt beide Arten.

6.2.3 Logische Fehler

Lösung: swap-Methode muss mit einem Datentyp realisiert werden, der die Übergabe durch Referenz umsetzt.

```
public void goodSwap (int[]feld) {
  int temp = feld[0];
  feld[0] = feld[1];
  feld[1] = temp;
}
```


6.2.4 Versteckte Fehler

Man kann relativ einfach feststellen ob ein Algorithmus syntaktisch korrekt ist, aber man kann nicht ohne Einschränkungen sagen, ob ein Algorithmus korrekt ist.

Dementsprechend kann es auch nach einer Programmprüfung versteckte Fehler geben.

Diese können nicht nur sehr kostspielig werden, man kann sie auch leider nur sehr schwer finden.

6.3 Debugging

Debugging bedeutet Fehler in einer Anwendung zu verbessern

Der Begriff „Debugging“ kommt aus einer Zeit, als Computer noch aus sehr vielen Röhren bestanden. Dummerweise haben sich Wanzen zwischen dem warmen Röhren wohl gefühlt. Manche Wanzen haben schon mal einen Kurzschluss verursacht – und somit Fehler. Debugging heißt als im ursprünglichem Sinne:

Entwanzen.

6.3 Die 9 Debugging-Regeln

1. Versteh das System (auch wenn es schwer fällt)
2. Reproduziere das Versagen (auch wenn es schwer fällt)
3. Nicht denken, hingucken! (selbst wenn du glaubst, du weißt was los ist)
4. Teile und herrsche (keine voreiligen Schlüsse)
5. Ändere immer nur eine Sache (selbst wenn sie trivial erscheint)

6.3 Die 9 Debugging-Regeln

6. Mach Notizen was passiert
7. Prüfe Selbstverständlichkeiten (zumindest nach einiger Zeit vergeblicher Suche)
8. Frag Außenseiter um Rat (zumindest nach einiger Zeit vergeblicher Suche)
9. Wenn du es nicht repariert hast, ist es auch nicht repariert (also reparier es und prüfe dann noch einmal nach)

6.3 Debugging

Herausfinden ab wann ein falscher Wert erzeugt wird!

Dazu verschieden Möglichkeiten.:

- Gedankliche Überprüfung aller Methoden und versuchte Anwendung der Bildungsvorschrift

System.out.println();

Breakpoints

... der/die Fehler kann/können überall sein!!!

6.3 Debugging

- **Oftmals:**

  „sinnlose“ Veränderung, die trotzdem das gewünschte Ergebnis liefert

► never change a running system ☺

6.3.1 Debugging durch System.out.println();

- **Ausgabe der relevanten Zwischenergebnisse:**

  so kann man nicht nur recht anschaulich erkennen wie das Programm arbeitet, sondern auch ab wann die Ausgabe falsch ist

6.3.2 Debugging durch Breakpoints

- **Breakpoint = vom Nutzer bestimmte Stelle an der das Programm die Ausführung unterbricht**

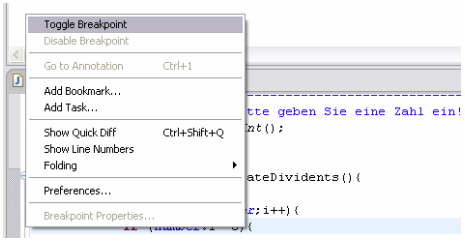
Somit hat der Benutzer dann die Möglichkeiten:

- Variablen mit ihren Werten beobachten
- das Programm schrittweise abzuarbeiten, bzw. Schritte zu überspringen

6.3.2.1 Breakpoints in Eclipse

Breakpoint setzen:

1. Rechtsklick an den Seitenrand des Programmfensters
2. „Toogle Breakpoint“ wählen

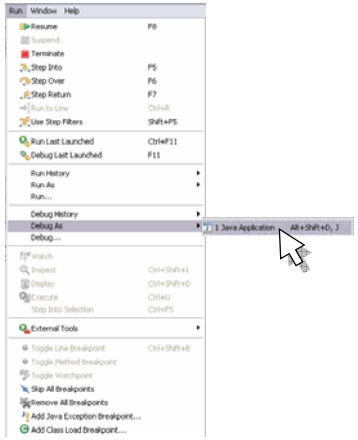


-> Breakpoint kann durch nochmaligen Rechtsklick und „Toogle Breakpoint“ gelöscht werden bzw. über „Disable Breakpoint“ inaktiv gemacht werden

6.3.2.1 Breakpoints in Eclipse

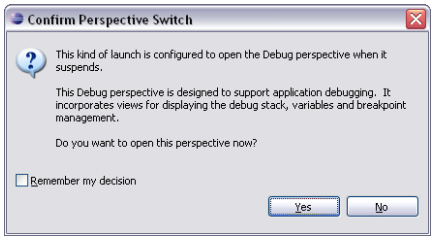
Nachdem die Breakpoints gesetzt wurden, muss das Programm nun wie folgt debugt werden:

-> „Run“ / „Debug As“ / „Java Application“



6.3.2.1 Breakpoints in Eclipse

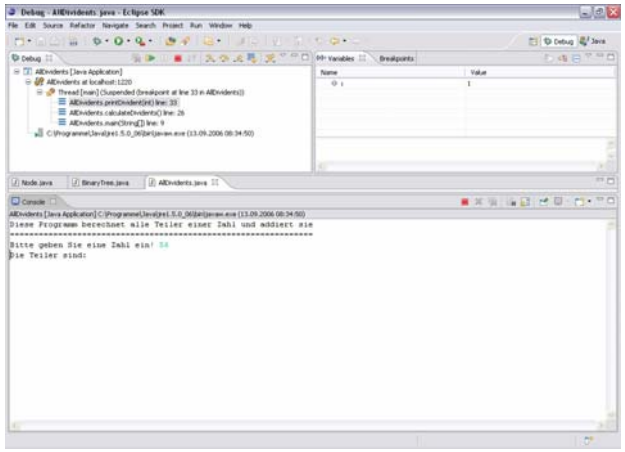
Eventuell öffnet sich dann folgende Meldung:



Diese bejahen und eventuell „Remember my decission“ klicken.
Das Programm wechselt dann in den Debug-Modus

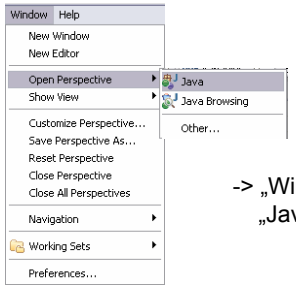
6.3.2.1 Breakpoints in Eclipse

Daraufhin öffnet sich die Debug-Ansicht:



6.3.2.1 Breakpoints in Eclipse

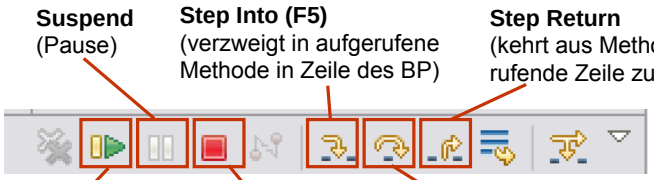
Wechseln in die Standard Java-Ansicht erfolgt folgendermaßen:



-> „Window“ / „Open Perspektive“ / „Java“

6.3.2.1 Breakpoints in Eclipse

Im Debug-Modus hat man dann folgende Möglichkeiten:



Suspend
(Pause)

Step Into (F5)
(verzweigt in aufgerufene Methode in Zeile des BP)

Step Return
(kehrt aus Methode in rufende Zeile zurück)

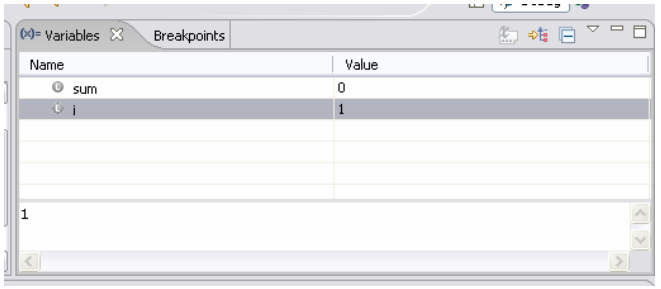
Resume
(Programm wird bis zum nächsten BP abgearbeitet)

Terminate
(beendet Programm)

Step Over (F6)
(überspringt Methodeaufruf)

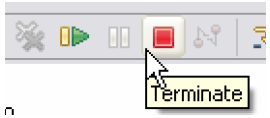
6.3.2.1 Breakpoints in Eclipse

Dabei kann man bei jedem Schritt die Werte der Variablen überprüfen:



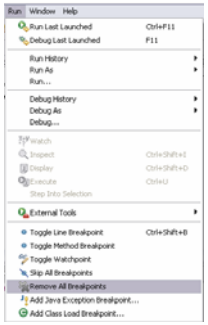
6.3.2.1 Breakpoints in Eclipse

Wenn Fehler behoben:



1. Debuggen beenden

2. Alle Breakpoints entfernen (eventuell)



-> „Run“ / „Remove All Breakpoints“

6.3 Debugging

Was tun wenn das alles nichts bringt?

- Pause machen
- alles Geschriebene bei Seite legen und noch einmal neu anfangen
- andere um Hilfe bitten bzw. surfen ;)