Einführung in die Informatik für Hörer aller Fakultäten II

~~Andreas Podelski~~

~~Stephan Diehl~~

Uwe Waldmann

Organisatorisches

Vorlesung:

Donnerstags, 14:15–16:00 Uhr, Geb. 45, Hörsaal 001.

Dozent:

Uwe Waldmann,
Gebäude 46.1 (MPI), Raum 627,
Tel.: (0681) 9325-227 bzw. 92227,
E-Mail: `uwe@mpi-sb.mpg.de`

Übungen:

Werner Backes,
Gebäude 46.1 (MPI), Raum 620,
E-Mail: `backes@mpi-sb.mpg.de`

Organisatorisches

Skript:

nein, aber Folienkopien auf Webseite

Vorlesungswebseiten:

`http://www.mpi-sb.mpg.de/~backes/einfo/einfo2.WS0001/`

`http://www.mpi-sb.mpg.de/~uwe/lehre/einfo2/`

Übungen:

ab nächste Woche

Scheinvergabe:

Übungsaufgaben + Klausur

Inhalt

Objektorientierte Programmierung

- wie programmiert man größere Progamme in Java?
- wie schreibt man verständliche Programme?
- wie schreibt man wartbare Programme?

Algorithmen und Datenstrukturen

- wie löst man bestimmte Probleme?
- wie löst man sie effizient (und was heißt hier „effizient“)?

Theorie

- welche Probleme kann man überhaupt mit einem Rechner lösen?

...

Literatur

Russell L. Shackelford: *Introduction to Computing and Algorithms.*
Addison-Wesley, 1998.

Ken Arnold, James Gosling: *The Java Programming Language.*
Addison-Wesley, 1997.

Mary Campione, Kathy Walrath: *The Java Tutorial.*
Addison-Wesley, 1998.

Bruce Eckel: *Thinking in Java.*
Prentice Hall PTR, 1998.

Ernst-Wolfgang Dieterich: *Java.*
R. Oldenbourg Verlag, 1999.

Vorausschau: die nächsten Wochen

Wiederholung Java:

Datentypen, Operationen,
Kontrollstrukturen, Funktionen, Rekursion,
Programmierstil

Konkretes Beispiel:

Problem: Suchen (eines Eintrags in einem Array)
Eine mögliche Lösung
Komplexität
Eine (etwas) bessere Lösung

Vorausschau: die nächsten Wochen

Strukturierung von Programmen:

Ziel: So programmieren, daß eine Lösung eines Teilproblems „schmerzlos" gegen eine andere ausgetauscht werden kann.
$\leadsto$ Objekte und Klassen

Ziel: So programmieren, daß Programmteile, die sich nur in Details unterscheiden, nicht komplett mehrfach programmiert werden müssen.
$\leadsto$ Klassenhierarchie

Konkretes Beispiel (Fortsetzung):

Noch bessere Lösungen (u.a.: verkettete Datenstrukturen)

Java – Wiederholung

Deklarationen

`int anzahl;`

`int`	Datentyp: Art der Daten (ganze Zahlen) und Wertebereich (-2^{31} ... $2^{31}-1$)
`anzahl`	Name der Variablen

Variable können sofort initialisiert werden:

```
int anzahl = 0;
```

Konstanten:

```
final int MONATE = 12;
```


(Primitive) Datentypen

Zahlen:

Ganzzahlig (mit Vorzeichen):

`byte`	[8bit]	(-128 ... 127)
`short`	[16bit]	(-32768 ... 32767)
`int`	[32bit]	(-2^{31} ... $2^{31}-1$)
`long`	[64bit]	(-2^{63} ... $2^{63}-1$)

Fließkommazahlen/„Reelle Zahlen“:

`float`	[32bit]	(ungefähr auf 10 Dezimalstellen genau)
`double`	[64bit]	(ungefähr auf 20 Dezimalstellen genau)

(Primitive) Datentypen

Wahrheitswerte:

`boolean` (`true`, `false`)

Zeichen:

`char` (`'A'`, `'z'`, `'*'`, `'\''`, `'\\'`, `'\n'`, ...)

Vorsicht mit „reellen Zahlen“

Bruch	Dezimal	Binär
1/2	0.5	0.1
3/4	0.75	0.11
1/3	0.333333 ...	0.01010101 ...

Mit endlich vielen Nachkommastellen nicht genau darstellbar

$\rightsquigarrow$ Rundungsfehler $\rightsquigarrow$ Gleichheitstest kann fehlschlagen

1/5	0.2	0.00110011 ...

Im Dezimalsystem mit endlich vielen Nachkommastellen genau darstellbar, aber im Binärsystem nicht!

$\rightsquigarrow$ unerwarteter Rundungsfehler

$\rightsquigarrow$ Kritisch bei kaufmännischen Anwendungen (DM 199.90)

Weitere Datentypen

Felder/Arrays:

```
int[] notenspiegel;

notenspiegel = new int[6];
notenspiegel[0] = 2;
notenspiegel[5] = 0;

n = notenspiegel.length;

notenspiegel = new int[] { 2, 6, 7, 4, 2, 0 };
```


Weitere Datentypen

Zeichenketten/Strings:

```
String name;

name = "Schulze";
name = "Andrea" + " " + name;
```

Objekte:

(demnächst mehr dazu).

Wichtiger Unterschied

bei primitiven Datentypen:

Variable enthält Wert:

```
int x, y;
...
if (x == y) {...}
  // Werte werden verglichen
```


Wichtiger Unterschied

bei anderen Datentypen:

Variable enthält Referenz/Verweis/Pointer auf Wert:

```
String x, y;
...
if (x == y) {...}
  // Referenzen werden verglichen:
  // kann false liefern,
  // obwohl x und y den gleichen Wert haben
...
if (x.equals(y)) {...}
  // Werte werden verglichen
```

Operatoren

arithmetisch:

`+`	Addition
`-`	Subtraktion
`*`	Multiplikation
`/`	Division (bei ganzen Zahlen: ganzzahliger Anteil)
`%`	Divisionsrest (Modulo-Operator)

Operatoren

Vergleich:

`==`	gleich
`!=`	ungleich
`>`	größer
`>=`	größer oder gleich
`<`	kleiner
`<=`	kleiner oder gleich

logisch:

`&&`	und
`||`	oder
`!`	nicht

Operatoren

Zuweisung:

`=`

kombinierte Zuweisung:

`+=`	`x += y` bedeutet `x = x + y`
`-=`	`x -= y` bedeutet `x = x - y`

inkr./dekr.:

`++`	`x++` oder `++x` bedeutet `x += 1`
`--`	`x--` oder `--x` bedeutet `x -= 1`

Kontrollstrukturen

if-else:

einfache Abfrage:

```
if (BEDINGUNG) {
  ANWEISUNGEN1
} else {
  ANWEISUNGEN2
}
```

else-Teil kann entfallen:

```
if (BEDINGUNG) {
  ANWEISUNGEN1
}
```

Kontrollstrukturen

if-else:

mehrere Abfragen hintereinander:

```
if (BEDINGUNG1) {
  ANWEISUNGEN1
} else if (BEDINGUNG2) {
  ANWEISUNGEN2
} else {
  ANWEISUNGEN3
}
```


Kontrollstrukturen

while:

Abfrage am Anfang:

```
while (BEDINGUNG) {
  ANWEISUNGEN
}
```

Abfrage am Ende:

```
do {
  ANWEISUNGEN
} while (BEDINGUNG)
```


Kontrollstrukturen

for:

```
for (ANWEISUNG1; BEDINGUNG; ANWEISUNG2) {
  ANWEISUNGEN
}
```

ist im wesentlichen äquivalent zu:

```
ANWEISUNG1
while (BEDINGUNG) {
  ANWEISUNGEN;
  ANWEISUNG2
}
```


Kontrollstrukturen

geschweifte Klammern können weggelassen werden, wenn nur eine einzige Anweisung drin steht (besser nicht!)

Funktionen

Aufgabe:

Programm in überschaubare Teile zergliedern,
Code-Wiederholung vermeiden

Funktionen

```
public static String wiederholeString(String str, int anz)
/* Zauberspruch:            public static
   Ergebnistyp:             String
   Name der Funktion:       wiederholeString
   Parameter der Funktion: str, anz
   Typen der Parameter:     String, int
   Wirkung:                 hänge str anz-mal hintereinander
*/
```


Funktionen

```
public static String wiederholeString(String str, int anz) {
  // Lokale Variable:
  String ergebnis;
  int i;

  ergebnis = "";
  for (i = 1; i <= anz; i++) {
    ergebnis = ergebnis + str;
  }

  // Funktionsaufruf beenden und ergebnis zurückgeben:
  return ergebnis;
}
```


Funktionen

```
public static String wiederholeString(String str, int anz) {
  if (anz == 0) {
    return "";
  } else {
    // rekursiver Aufruf von wiederholeString:
    return wiederholeString(str, anz - 1) + str;
  }
}
```

Funktionen

falls nichts zurückgegeben werden soll: Ergebnistyp void

```
public static void druckeString(String str) {
  // ... irgendwelche Seiteneffekte ...
}

public static void macheNichts() {
}
```


Programmierstil

Aussagekräftige Namen:

- mnemonisch, nicht zu kurz:
  - umso wichtiger, je seltener der Name verwendet wird und je weiter verstreut die Stellen sind, an denen er verwendet wird.
  - (`i,j,k` für Schleifenzähler ist in Ordnung.)
- nicht zu lang:

```
groesstesElementDerErstenListe =
      groesstesElementDerErstenListe + 1;
```


Programmierstil

Namenskonvention:

- Variablennamen, Funktionsnamen:
  - kleiner Anfangsbuchstabe, danach groß und klein gemischt
- Konstanten:
  - nur Großbuchstaben
- demnächst: Klassen:
  - großer Anfangsbuchstabe, danach groß und klein gemischt

Programmierstil

Einrückungen:

- Anweisungen im Inneren einer Schleife oder Verzweigung werden grundsätzlich gegenüber den äußeren Anweisungen eingerückt.
- Einrückung: mindestens 2 Leerzeichen, höchstens 8 Leerzeichen, (aber einheitlich!)