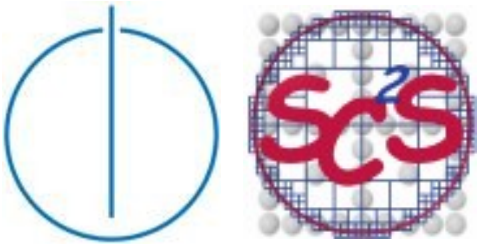
Einführung in die Programmierung I

8. Prozedurale Konzepte in Java, Teil1

Stefan Zimmer

17.12.2007



Von Maple zu Java

- Maple ist für größere Projekte kaum die geeignete Sprache
- Daher wenden wir das Gelernte an, um noch eine weitere Programmiersprache zu erlernen
- Hier wird das Java sein – für andere Sprachen wäre das aber im Grunde dasselbe: wenn wir die Konzepte verstanden haben, sollten wir in der Lage sein, in kurzer Zeit die Sprache X oder Y oder Z zu lernen.

Standardtour „Neue Programmiersprache“

- Beim Lernen einer neuen Programmiersprache kann man sich z.B. folgende Gedanken machen:
  - Was für *Datentypen* sieht die Sprache vor?
  - Wie stelle ich *Ausdrücke* dar?
  - Wie kann ich *Variablen* vereinbaren?
  - Wie sehen die Standardanweisungen aus (*Zuweisung*, *Fallunterscheidung*, *Schleife*)?
  - Wie kann ich Programmteile zusammenfassen und wiederverwenden (*Prozeduren* etc.)?

Wir lernen Java

- Die Fragen der vorigen Folie werden uns die nächsten zwei Wochen beschäftigen („Prozedurale Konzepte“, Teil 1 und 2).
- Die übrigen beiden Wochen lernen wir eine Technik, die wir von Maple nicht kennen: die *objektorientierte Programmierung*.
- Die ist ein so zentraler Bestandteil von Java, dass unsere in anfänglicher Unkenntnis geschriebenen Programme etwas merkwürdig aussehen werden, die Gründe dafür werden später hoffentlich klar werden.

Material

- Den Übersetzer, der aus einem Java-Programm einen ausführbaren Code erzeugt, gibt es bei SUN kostenlos:
  `http://java.sun.com/j2se`
- Die technischen Unschönheiten kann man hinter schönen bunten Benutzeroberflächen verstecken, für unsere (Lern-) Zwecke gut geeignet (und umsonst) ist BlueJ (näheres dazu morgen in der Übung):
  `http://www.bluej.org/`
- Wie bei Maple gilt auch hier: eine ältere Version reicht völlig aus.

Literatur

- Auf den Java-Seiten von SUN gibt es ein schönes Tutorial, das als begleitende Lektüre völlig ausreicht (Teile davon, natürlich!).
- Bücher über Java gibt's viele, zwei davon seien hier erwähnt:
  - *R. Schiedermeier: Programmieren mit Java, Pearson Studium*
    Ähnlicher Aufbau wie unsere Betrachtungen: von den grundlegenden Dingen hin zur objektorientierten Programmierung
  - *D. Barnes, M. Kölling: Objektorientierte Programmierung mit Java, Pearson Studium*
    Ein Vertreter des komplementären Ansatzes: man kann auch mit den Objekten anfangen und sich zu den Details vorarbeiten

Was nicht behandelt wird

- Wir kümmern uns hier um die Sprache Java, trotzdem sei erwähnt, dass zu den Vorzügen der Sprache die große Menge (standardisierter) Programmbibliotheken gehört
- Damit lassen sich z.B. schöne bunte Benutzeroberflächen zusammenbauen und Internet-Business betreiben (in dem Zusammenhang hört man ja auch öfter mal was von Java)
- Bestandteil der Sprache ist das alles aber nicht.

Datentypen

- Wichtiges Prinzip in Java: jede Variable hat einen festen *Datentyp*, der angibt, was für eine Sorte von Daten darin gespeichert werden kann (das ist in den meisten Programmiersprachen so, in Maple aber nicht).

- Bevor wir lernen, wie man der Variablen den Datentyp zuordnet, erstmal ein Katalog von in Java vorgesehenen Datentypen

- Grundlegende Unterscheidung in Java: *primitive Datentypen* (heute) und *Referenzdatentypen* (später)

Ganzzahlige Datentypen

- Für Variablen, die ganzzahlige Werte speichern sollen; unterschiedlicher Wertebereich
- **`int`** (der Standardtyp für ganze Zahlen): -2^{31} ...$2^{31}-1$
- Weniger Speicherplatz: **`byte`** (-2^7...2^7-1), **`short`** (-2^{15}...$2^{15}-1$)
- Mehr Speicheplatz: **`long`** (-2^{63}...$2^{63}-1$)
- Literale (Typ `int`) wie gewohnt (`1234`, `-5` etc.)

Fließkomma-Datentypen

- Zahlen mit Nachkommastellen und größerem Wertebereich (aber Vorsicht: Rundung!)
- **`double`** (der Standardtyp für Fließkommazahlen): 8 Byte, etwa 15 Dezimalstellen
- **`float`**: 4 Byte, etwa 7 Dezimalstellen
- Literale mit Dezimalpunkt und/oder Exponent (sonst: `int`):
  `12.34, 0.1234E2, 1234E-2`
  sind drei Schreibweisen für dieselbe Zahl.

Weitere primitive Datentypen

- **`char`**: zum Speichern eines Zeichens (Buchstabe, Ziffer etc.)
  Literale in Hochkommas, z.B.:
  - **`'a'`**, **`'A'`**, **`'z'`**, **`'Z'`**, **`'ä'`**
  - **`'7'`** (die Ziffer, nicht die Zahl `7` oder die Zahl `7.0`)
  - **`'\n'`** (Zeilenumbruch – *ein* Zeichen!)

- **`boolean`**: für Wahrheitswerte; vollständige Liste der Literale: **`true`** und **`false`**

Ausblick: Referenzdatentypen

- Der Gegensatz zu „primitiver Datentyp“ ist ja „Referenzdatentyp“ – darunter werden alle weiteren Datentypen fallen, die wir (später) kennen lernen werden: insbesondere Felder und Klassen

- Zeichenketten (Strings) fallen auch darunter, vorab lernen wir, dass String-Literale Gänsefüßchen tragen:
  - `"Hallo"`
  - `"a"` (verschieden vom Zeichen `'a'`)
  - `"x\ny"` (String aus 3 Zeichen: `x`, Zeilenumbruch, `y`)

Bezeichner, Deklarationen

- *Bezeichner* bestehen aus einem Buchstaben, optional gefolgt von weiteren Buchstaben oder Zahlen (Java-Schlüsselwörter sind verboten):
  `x, x1, ziemlichLangerBezeichner`

- Die Verknüpfung des Bezeichners mit dem Datentyp erfolgt mit einer *Deklaration*; diese enthält den Typnamen, eine durch Kommas getrennte Liste der Bezeichner und ein Semikolon:

```
int x;
double zahl1, zahl2, zahl3;
```

- So eine Deklaration als Anweisung nennt man Variablendeklaration, durch sie wird auch die passende Menge Speicherplatz bereit gestellt.

Ausdrücke

- Aus Literalen und Bezeichnern können wir wie in Maple Ausdrücke (Formeln) zusammensetzen
- Anders als in Maple ist eine Formel hier kein Objekt, das man z.B. in einer Variable speichen könnte, sondern nur eine Rechenvorschrift: wenn das Programm abläuft, werden die Formeln (vollständig) ausgewertet
- Für die Reihenfolge der Auswertung von Ausdrücken wie `3*x+4*y` gibt es wieder Vorrangsregeln, ggf. Klammern `()` setzen

Arithmetische Ausdrücke

- Es gibt die arithmetischen Operatoren:
  `+, -, * , /` und `%` (Modulo: `11%3` gibt `2`)
- Vorsicht: die Operatoren stehen je nach Typ der Operanden für ganz unterschiedliche Operationen: `5.0/2.0` gibt `2.5` (Gleitkomma-Division) `5/2` gibt `2` (Ganzzahlige Division)
- Und wenn wir Datentypen mischen? Dann wandelt Java gemäß der Anordnung

  `byte` → `short` → `int` → `long` →

  `float`→ `double`

  den „kleineren“ Datentyp in den „größeren“ um (ggf. mit Rundung): `5/2.5` gibt den double-Wert `2.0`

Vergleichsoperatoren

- Die Operatoren **<**, **<=**, **>** und **>=** führen, angewendet auf Zahlen oder Zeichen, die üblichen Vergleiche aus.

- Vorsicht: der Test auf Gleichheit heißt in Java **==** (das einfache = wird der Zuweisungsoperator werden), das „Ungleich“ ist **!=**

- Ergebnistyp aller dieser Operatoren ist `boolean`

Logische Operatoren

- Es gibt die üblichen Operationen für Wahrheitswerte (`x` und `y` seien Ausdrücke mit Ergebnistyp `boolean`):
- `x && y` ist genau dann `true`, wenn `x` und `y` `true` ergeben, sonst `false`
  (ergibt `x false`, wird `y` nicht ausgewertet)
- `x || y` ist genau dann `false`, wenn `x` und `y false` ergeben, sonst `true`
  (ergibt `x true`, wird `y` nicht ausgewertet)
- `!x` ist genau dann `true`, wenn `x false` ist und umgekehrt

Zuweisungen

- Zuweisungen finden mittels = statt: das Ergebnis der rechten Seite (ein beliebiger Ausdruck) wird der linken Seite zugewiesen (derzeit muss das der Name einer Variablen passenden Typs sein):

```
int a, b;
a = 5;
b = a/2 + 2;
```

(nun enthält `a` den Wert 5, `b` den Wert 4)

- Eine Zuweisung mit Semikolon ist eine Anweisung

- Auch erlaubt: in der Variablendeklaration gleich einen Wert zuweisen:

```
int a = 5, b;
b = a/2 + 2;
```

Sonstige Operatoren

- Die Operatoren `+` und `-` können auch als Vorzeichen auftreten: `+4, -5`
- Der Operator `++` hinter einem Variablennamen erhöht die Variable um 1 und gibt den ursprünglichen Wert zurück:
  ```
  int a,b; a = 4; b = a++;
  ```
  (nun enthält `a` den Wert 5, `b` den Wert 4)
- Analog erniedrigt `--` den Wert um 1:
  ```
  int a,b; a = 4; b = a--;
  ```
  (nun enthält `a` den Wert 3, `b` den Wert 4)
- Ein Inkrement- (`++`) oder Dekrement- (`--`) Ausdruck mit Semikolon ist auch eine Anweisung: `a++;` oder `b--;`

Kommentare

- Kommentare (Text, der vom Übersetzer ignoriert wird, aber hoffentlich die Lesbarkeit des Programms erhöht) gibt es in zweieinhalb Formen:
- *Zeilenkommentare* beginnen mit `//` und umfassen den ganzen Rest der Zeile:

```
a = 1; // Ein (überflüssiger) Kommentar
```

- *Blockkommentare* beginnen mit `/*` und enden mit `*/` (sie dürfen also über mehrere Zeilen gehen):

```
/* Erste Zeile des Kommentars
   Zweite Zeile
*/
```

- Eine besondere Form von Blockkommentaren beginnt mit `/**` und heißt dann *Doc-Kommentar*: wenn man deren Inhalt passend füllt, kann mittels des Programms javadoc daraus eine schöne Dokumentation erstellt werden

(Klassen-) Methoden

- Den Maple-Prozeduren entsprechen in Java die Methoden, genauer gesagt die *Klassenmethoden* (später werden wir auch *Objektmethoden* kennen lernen)
- Betrachten wir folgende Maple-Prozedur:

```
blub := proc(a::integer; b::integer)::integer
  local c;
  c := a*b + 5;
  return c
end proc;
```

- In Java könnte das etwa so aussehen:

```
static int blub(int a, int b) {
    int c;
    c = a*b + 5;
    return c;
}
```

Methodendeklaration: Bestandteile

- Das Schlüsselwort `static` („Klassenmethode“)
- Datentyp des Funktionsergebnisses (bzw., falls kein Ergebnis anfällt: `void`)
- Methodenname (ein Bezeichner)
- Liste der Formalparameter: kein oder mehr Paare `Datentyp Bezeichner`, durch Komma getrennt in runden Klammern
- Dann der *Methodenrumpf*: ein Paar geschweifter Klammern und darin Anweisungen (incl. Variablendeklarationen); Ergebnisrückgabe mittels `return` (entfällt im Fall `void`, sonst Pflicht)

Methoden verwenden

- Der Aufruf einer Methode sieht aus wie in Maple: der Methodenname, gefolgt von den Aktualparametern – eine Liste von Ausdrücken, durch Kommas getrennt und in runden Klammern: `blub(5,7)`
- Ein Methodenaufruf kann Bestandteil eines Ausdrucks sein; mit Semikolon ist er auch eine vollwertige Anweisung.
- Die Wirkung ist dieselbe wie in Maple:
  - die Werte der Aktualparameter werden berechnet,
  - sie werden für die entsprechenden Formalparameter eingesetzt,
  - die Anweisungen im Methodenrumpf werden ausgeführt,
  - ggf. wird mit dem Ergebnis weitergerechnet.

Programmrahmen

- Damit aus unsern Methoden ein korrektes Java-Programm wird, muss noch ein Rahmen drumrum, der so aussieht (warum, lernen wir später):

```
public class Klassenname {
// hier die Klassenmethoden einfügen
}
```

- *Klassenname* ist dabei ein beliebiger Bezeichner, den wir auch als Name der Datei verwenden; die Datei bekommt die Endung `.java`, heißt im Beispiel also `Klassenname.java`
- Die kann man mit einem beliebigen Editor erstellen und in den Java-Übersetzer stecken.

Programm ausführen

- In der (empfohlenen) Weichei-Variante mit BlueJ kann man nun mittels Mausklick die Methoden anstoßen (da kriegt man den Rahmen auch gleich vorgefertigt).
- Wenn man das Programm „von Hand“ mittels der Programme `java` oder `jre`) starten will, muss man eine Klassenmethode
  `public static void main(String [] args)`
  deklarieren, bei der die Programmausführung gestartet wird.

Anweisungen: Blöcke

- Nun fehlen uns noch die Kontrollstrukturen – vorher lernen wir aber noch, was ein *Block* ist, nämlich eine in geschweifte Klammern eingeschlossene (möglicherweise leere) Folge von Anweisungen.
- Ein Block selber ist *eine* Anweisung.
- In einem Block deklarierte Variablen existieren nur innerhalb dieses Blocks – außerhalb ist der Bezeichner undefiniert; der Speicherplatz wird bei Verlassen des Blocks freigegeben.
- Analoges gilt für die Variablen, die im Methodenrumpf (und nicht innerhalb untergeordneter Blöcke) deklariert sind und für die Formalparameter.

Fallunterscheidung

- Die Fallunterscheidung mit `if` kann in zwei Formen auftreten:
  `if` **`(`***`Bedingung`***`)`** *`Anweisung`*
  oder mit else-Zweig:
  `if` **`(`***`Bedingung`***`)`** *`Anw1`* **`else`** *`Anw2`*

- Die runden Klammern sind Bestandteil des `if`

- Die Anweisungen sind in der Regel Blöcke, die geschweifte Klammern mitbringen, die gehören aber nicht zum `if`. Hier z.B. sind die Klammern optional:

```
if (x > 0) {
    z = y/x;
} else {
    z = 0;
}
```

while-Schleifen

- Die `while`-Schleife sieht ganz ähnlich aus wie in Maple:
  `while` **`(`***`Bedingung`***`)`** *`Anweisung`*
- Sie funktioniert auch genau so, wie wir das erwarten: die Anweisung (in der Regel wieder ein Block) wird wiederholt, solange die Bedingung `true` ergibt.
- Es gibt noch eine Variante, bei der die Bedingung nach Ausführen der Anweisung überprüft wird:
  `do` *`Anweisung`* **`while`** **`(`***`Bedingung`***`);`**

for-Schleifen

- Die for-Schleife in Java ist vom Grundgedanken der for-Schleife (Zahl der Schleifendurchläufe ist von Beginn an bekannt) noch weiter entfernt als die in Maple. Aber sie ist praktisch!
- So sieht sie aus:

```
for (Start; Bedingung; Weiter)
     Anweisung
```

- Das ist eine bequeme Notation für Folgendes:

```
{    Start;
     while (Bedingung) {
          Anweisung
          Weiter;
     }
}
```

for-Schleife: Beispiel

- Sehen wir uns das für einen Standardfall an, z.B. von 1 bis 10 zählen:
  - „Start“ (erlaubt ist ein Ausdruck – streng genommen: ein Ausdruck, der auch eine Anweisung bilden könnte – oder eine Variablendeklaration ohne ;): `int i = 1`
  - „Bedingung“ (ein Ausdruck mit Ergebnistyp `boolean`): `i<=10`
  - „Weiter“ (ein Ausdruck, Einschränkung vgl. „Start“): `i++` (oder `i=i+1`)
  - „Anweisung“, hier als Platzhalter der Methodenaufruf `tuWas(i);`
- Im ganzen Satz:

```
for (int i=1; i<=10; i++) tuWas(i);
```

Standardmethoden zur Ausgabe

- Damit wir irgendwas auf den Bildschirm bekommen: es sind für primitive Datentypen Methoden `System.out.println` mit einem Parameter vordefiniert, die den Wert dieses Parameters auf dem Bildschirm ausgibt, z.B. `System.out.println(123.456);`

- Es ist in Java erlaubt, dass man mehrere Methoden mit gleichem Namen hat, solange sie sich in den Datentypen ihrer Parameter unterscheiden – diesen Sachverhalt nennt man *Überladung*.