Kapitel 6: Prozedurale Konzepte in Java i

Kapitel 6: Prozedurale Konzepte in Java ii

Variablen und Konstanten in Java

- Eine Variablendeklaration hat in Java die Gestalt:

Syntaxregel (Variablendeklaration)

⟨*VarDeklaration*⟩ ::= ⟨*TypName*⟩ ⟨*varName*⟩`;`

- Konvention: Variablennamen beginnen mit kleinen Buchstaben.
- Beispiel: `int x;`

Variablen und Konstanten in Java

- Eine Konstantendeklaration hat in Java die Gestalt:

Syntaxregel (Konstantendeklaration)

⟨*KonstDeklaration*⟩ ::= **`final`** ⟨*TypName*⟩ ⟨*konstName*⟩**`;`**

- Konvention: Konstantennamen bestehen komplett aus großen Buchstaben.
- Beispiel: `final int PI;`
- Auch in Java hat jede Variable/Konstante damit einen Typ (eine Deklaration ist also das Bereitstellen eines Platzhalters des entsprechenden Typs).

Variablen und Konstanten in Java

- Wie bereits erwähnt, müssen Variablen/Konstanten in einem Java Programm nicht am Anfang einer Methode vereinbart werden, sondern können auch “zwischendurch” eingeführt werden (überall dort, wo eine Anweisung stehen darf, da es sich dabei auch um eine Anweisung handelt).
- Das bedeutet, dass die Menge $N(\mathcal{S})$ während einer Methode “wachsen” kann.
- Der Compiler hält beim Übersetzen in Bytecode die aktuelle Liste, um die (syntaktische) Gültigkeit von Ausdrücken überprüfen zu können.
- Bezeichner, die erst später im Code deklariert werden, können also vorher nicht in Ausdrücken verwendet werden.

Variablen und Konstanten in Java

- Eine Wertzuweisung hat die Gestalt:

Syntaxregel (Wertzuweisung)

⟨*Wertzuweisung*⟩ ::= ⟨*varName*⟩|⟨*konstName*⟩ = ⟨*Ausdruck*⟩ **;**

- Eine Variablen- bzw. Konstantendeklaration kann auch mit der Initialisierung verbunden sein, d.h. der ersten Wertzuweisung:

Syntaxregel (Initialisierung)

⟨*Initialisierung*⟩ ::= {**final**}* ⟨*TypName*⟩ ⟨*varName*⟩|⟨*konstName*⟩ = ⟨*Ausdruck*⟩ **;**

- Wie gesagt, Deklaration, Initialisierung und Wertzuweisung sind *Anweisungen*, die wie bei uns mit einem Semikolon beendet werden.

Beispiele

- Konstanten: `final <typ> <NAME> = <ausdruck>;`

```
final double Y_1 = 1;
final double Y_2 = Y_1 + 1 / Y_1;
final double Y_3 = Y_2 * Y_2;
final char NEWLINE = '\n';
final double BESTEHENSGRENZE_PROZENT = 0.5;
final int GESAMTPUNKTZAHL = 80;
```

Beispiele

- Variablen: `<typ> <name> = <ausdruck>;`

```
double x = 7.43;
double y = x + 1;
int klausurPunkte = 42;
boolean klausurBestanden =
      ((double) klausurPunkte) /
        GESAMTPUNKTZAHL >=
           BESTEHENSGRENZE_PROZENT;
```

Abkürzungen für Wertzuweisungen in Java

Für bestimmte einfache Operationen (Addition und Subtraktion mit 1 als zweitem Operanden) kennen wir schon Kurznotationen:

Operator	Bezeichnung	Bedeutung
`++`	Präinkrement	`++a` ergibt `a+1` und erhöht `a` um `1`
`++`	Postinkrement	`a++` ergibt `a` und erhöht `a` um `1`
`--`	Prädekrement	`--a` ergibt `a-1` und verringert `a` um `1`
`--`	Postdekrement	`a--` ergibt `a` und verringert `a` um `1`

Also steht z.B. `a++;` für `a = a + 1;`. Besonders ist, dass es sich dabei um Ausdrücke handelt, die einen Wert haben.

Abkürzungen für Wertzuweisungen in Java

Wenn man eine Variable nicht nur um 1 erhöhen oder verringern, sondern allgemein einen neuen Wert zuweisen will, der aber vom alten Wert abhängig ist, gibt es Kurznotationen wie folgt:

Operator	Bedeutung
`+=`	`a+=b` weist `a` die Summe von `a` und `b` zu
`-=`	`a-=b` weist `a` die Differenz von `a` und `b` zu
`*=`	`a*=b` weist `a` das Produkt aus `a` und `b` zu
`/=`	`a/=b` weist `a` den Quotienten von `a` und `b` zu

(i.Ü auch für weitere Operatoren möglich...)

Anweisungen

- Wie wir bereits festgestellt haben, steht eine Anweisung für einen einzelnen Abarbeitungsschritt in einem Algorithmus.
- Einige wichtige Arten von Anweisungen in Java sind:
  - Die *leere* Anweisung, bestehend aus einem einzigen `;`
  - Vereinbarungen und Initialisierungen von Variablen/ Konstanten wie oben.
  - *Ausdrucksanweisung*: `<ausdruck>;`
    Dabei spielt der Wert von `<ausdruck>` keine Rolle, die Anweisung ist daher nur sinnvoll (und in Java nur dann erlaubt), wenn `<ausdruck>` einen Nebeneffekt hat, z.B.
    - Wertzuweisung von Variablen/Konstanten,
    - Funktions-/Prozeduraufruf (werden wir später kennenlernen),
    - Instanzerzeugung (werden wir später kennenlernen)

Der Anweisungsblock

- Man kann eine Menge von Anweisungen zu einem *Block* zusammen fassen.
- Ein Block beginnt in Java mit einer öffnenden geschweiften Klammer und endet mit einer schließenden geschweiften Klammer, die eine beliebige Menge von Anweisungen umschließen:

```
{
  Anweisung1;
  Anweisung2;
  ...
}
```

Der Anweisungsblock

- Die Anweisungen im Block werden nacheinander ausgeführt.
- Der Block als Ganzes gilt als eine einzige Anweisung, kann also überall da stehen, wo syntaktisch eine einzige Anweisung verlangt ist.
- Blöcke können beliebig geschachtelt sein.
- Der Rumpf einer Methode bildet demnach ebenfalls einen Block.

Lebensdauer, Gültigkeit, Sichtbarkeit

- Das Konzept des (Anweisungs-)Blocks ist wichtig im Zusammenhang mit Variablen und Konstanten.
- Die *Lebensdauer* einer Variablen ist die Zeitspanne, in der die virtuelle Maschine der Variablen einen *Speicherplatz* zu Verfügung stellt.
- Die *Gültigkeit* einer Variablen erstreckt sich auf alle Programmstellen, an denen der Name der Variablen dem Compiler durch eine Deklaration bekannt ist.
- Die *Sichtbarkeit* einer Variablen erstreckt sich auf alle Programmstellen, an denen man über den Namen der Variablen auf ihren Wert zugreifen kann.

Gültigkeitsbereich von Variablen

- Eine in einem Block deklarierte (*lokale*) Variable ist ab ihrer Deklaration bis zum Ende des Blocks gültig und sichtbar.
- Mit Verlassen des Blocks, in dem eine Variable lokal deklariert wurde, endet auch ihre Gültigkeit und Sichtbarkeit.
- Damit oder kurz danach endet normalerweise auch die Lebensdauer der Variablen, da der Speicherplatz, auf den die Variable verwiesen hat, im Prinzip wieder freigegeben ist (dazu später mehr) und für neue Variablen verwendet werden kann.
- Solange eine Variable sichtbar ist, darf keine neue Variable gleichen Namens angelegt werden.

Beispiel

```
...
int i = 0;
{
  int i = 1;   // nicht erlaubt, i ist schon bekannt
  i = 1;       // erlaubt
  int j = 0;
}
j = 1;         // nicht moeglich, j ist nicht bekannt
...
```

- Bei verschachtelten Blöcken: Variablen des äußeren Blocks sind im inneren Block sichtbar.

Prozeduren in Java

- In Java werden Prozeduren (wie Funktionen) durch *Methoden* realisiert.
- Eine Methode wird definiert durch den Methodenkopf (die Signatur der Methode) und den Methodenrumpf (bestehend aus einer Menge von Anweisungen in Block-Klammern).

Syntaxregel (Methodenkopf)

⟨*Methodenkopf*⟩ ::= **`public static`** ⟨*TypName*⟩ ⟨*MetName*⟩ (⟨*ParamListe*⟩)

Dabei ist

- ⟨*TypName*⟩ der Bildbereich (Ergebnistyp),
- ⟨*MetName*⟩ der Name der Methode,
- ⟨*ParamListe*⟩ spezifiziert die Eingabeparameter und –typen und besteht aus endlich vielen (auch keinen) Paaren von Typen und Variablennamen, die jeweils durch Komma getrennt sind,

Beispiel

```
public static int mitte(int x, int y, int z) {
   ...
   <Anwesiungen>
   ...
}
```

Prozeduren in Java

- Als besonderer Ergebnis-Typ einer Methode ist auch **`void`** möglich. Dieser Ergebnis-Typ steht für die leere Menge als Bildbereich.
- Eine Methode mit Ergebnistyp **`void`** gibt *kein* Ergebnis zurück; der Sinn einer solchen Methode liegt also ausschließlich in den Nebeneffekten.
- Das Ergebnis einer Methode ist der Ausdruck nach dem Schlüsselwort **`return`**; nach Auswertung dieses Ausdrucks endet die Ausführung der Methode.
  - Eine Methode mit Ergebnistyp **`void`** hat entweder keine oder eine leere **`return`**-Anweisung.
  - Eine Methode, die einen Ergebnistyp `<type>` $\neq$ **`void`** hat, muss mindestens eine **`return`**-Anweisung mit einem Ausdruck vom Typ `<type>` haben.

Prozeduren in Java

- Analog können wir nun auch das Modulkonzept erweitern, um benutzereigene Prozeduren bereitzustellen.
- Beispiel:

```
public class BewegungI
{
  /**
   * ...
   */
  public static double streckeI(double m, double t, double k)
  {
    double b = k / m;
    return 0.5 * b * Math.pow(t,2);
  }

   ...
}
```

Prozeduren in Java

- Eine Methode mit Ergebnistyp `<type>` $\neq$ **`void`** ist ein Ausdruck vom Typ `<type>` und kann überall dort stehen, wo ein Ausdruck vom Typ `<type>` verlangt ist (z.B. bei einer Wertzuweisung an eine Variable vom Typ `<type>`).
- Beispiel: `double s = streckeI(3.0, 4.2, 7.1);`
- Achtung: da Java nicht zwischen Funktion und Prozedur unterscheidet, könnte die Methode `streckeI` Nebeneffekte haben!

Prozeduren in Java

- Eine Methode mit Ergebnistyp `void` ist ein Ausdruck vom Typ `void` (ein Aufruf ist eine Ausdrucksanweisung).
- Die Seiteneffekte der Methode ist das (hoffentlich) beabsichtigte Resultat der Anweisung.
- Unser theoretisches Zustandsmodell gilt analog für Java Anweisungen und Prozeduraufrufe, d.h. Nebeneffekte bei Prozeduren sind mit unseren bisherigen Konzepten zunächst ohne Wirkung (call-by-value!)

Beispiele

- `System.out.println` ist eine Methode mit Ergebnistyp **`void`** und hat als Seiteneffekt: gib den Eingabeparameter (Typ `String` für Zeichenketten) auf dem Bildschirm aus. Entsprechend ist `System.out.println(...);` eine Ausdrucksanweisung.
- Die Inkrement- und Dekrement-Operatoren (z.B. `a++` sind Methoden mit Ergebnistyp (typw. **`int`**) haben aber wie besprochen als Seiteneffekte die Erhöhung/Erniedrigung des Arguments um 1.
  Entsprechend ist z.B. `a++;` eine Ausdrucksanweisung.

Prozeduren in Java

- Der Rumpf einer Methode definiert einen Anweisungsblock (daher auch die Blockklammern).
- Es gelten die bereits bekannten Regeln zu Lebensdauer, Sichtbarleit, etc. für Blöcke.
- Zusätzlich zu den lokalen Variablen, die innerhalb des Blocks vereinbart werden, sind noch die formalen Eingabe-Variablen der Methode sichtbar (so sind wir es ja auch gewohnt).

Prozeduren in Java

- Java setzt wie erwähnt call-by-value um:
- Beispiel:

```
public class Exchange
{
      public static void swap(int i, int j) {
            int c = i;
            i = j;
            j = c;
      }

      public static void main(String[] args) {
            int x = 1;
            int y = 2;
            swap(x,y);
            System.out.println(x); // Ausgabe?
            System.out.println(y); // Ausgabe?
      }
}
```

Prozeduren in Java

- Aufruf von `main`:
  - Zunächst werden nacheinander $(\texttt{x}, 1), (\texttt{y}, 2)$ angelegt (wir ignorieren `args` im Kopf von `main`).
  - Beim Aufruf von `swap` werden neue Zettel angelegt: `i` als Kopie für `x` und `j` als Kopie für `y`:
    $(\texttt{i}, 1), (\texttt{j}, 2)$
  - Dann
    $(\texttt{i}, 1), (\texttt{j}, 2), (\texttt{c}, 1)$
    $(\texttt{i}, 2), (\texttt{j}, 2), (\texttt{c}, 1)$
    $(\texttt{i}, 2), (\texttt{j}, 1), (\texttt{c}, 1)$
  - Alles schön und gut, aber nach dem Aufruf von `swap` in `main` (beim Ausgabe mit `System.out`) hat `x` den Wert 1 und `y` den Wert 2, d.h. `swap` hat keinerlei Wirkung auf `x` und `y` in `main`

Und das ist genau das, was wir formalisiert haben!

Prozeduren vs. Funktionen

- Mit der Formalisierung wird auch der Unterschied zwischen Prozeduren und Funktionen noch einmal klar.
- Der Aufruf beider Varianten bewirkt zunächst das Gleiche: die Eingabevariablen werden mit konkreten Werten belegt.
- Bei Funktionen wird anschließend der Wert des Rumpfes der Funktion (ein Ausdruck) ausgewertet; der Funktionsaufruf ist ja selbst wieder ein Ausdruck (mit einem Wert).
- Dabei wird die Variablenbelegung innerhalb des Rumpfes der Funktion nicht verändert, wir haben keine Zustandsänderung (daher hatten wir auch keinen Zustandsbegriff benötigt).

Prozeduren vs. Funktionen

- Bei Prozeduren können weitere Variablen und Konstanten (mit entsprechenden Belegungen) zu der Menge der Eingabevariablen hinzukommen.
- Die Anweisungen im Rumpf verändern ggf. deren Belegungen (das sind die vielbeschworenen Seiteneffekte), was wir mit dem Zustandsbegriff modelliert haben.

Prozeduren vs. Funktionen

- D.h. Im funktionalen Paradigma spezifiziert der Algorithmus, wie das Ergebnis aussieht (als Ausdruck), aber nicht wie es vom Computer berechnet werden soll (die Auswertung des Ausdrucks).
- Im imperativen Paradigma spezifiziert der Algorithmus die Berechnung des Ergebnisses (in einzelnen Schritten).
- Bei der (in Java nicht seltenen) Mischform [10] stellt der Ausdruck nach der `return`-Anweisung also nicht notwendigerweise den direkten Zusammenhang zwischen Eingabewerten und Ausgabe dar, da der Zustand am Ende des Methodenrumpfes typischerweise nicht mehr mit dem Startzustand (bei Aufruf) übereinstimmt.

[10] Prozeduren die einen Rückgabewert vom Typ $\mathtt{T} \neq$ `void` besitzen

Globale Größen

- Unsere bisherigen Variablen und Konstanten sind *lokale Größen*, die nur innerhalb des Blocks, der sie vereinbart, bekannt sind.
- Es gibt auch *globale* Variablen und Konstanten, die in mehreren Algorithmen und Modulen bekannt sind.
- Diese globalen Größen sind z.B. für den Datenaustausch zwischen verschiedenen Algorithmen verwendbar[11].
- Sie können in einem Modul (Klassenvereinbarung) spezifiziert werden und (wenn sie als `public` deklariert wurden) von den Methoden aller anderen Module (Klassen) verwendet werden.

[11] Aber Achtung: das widerspricht eigentlich gutem oo Stil — sh. später!

Globale Größen

- Globale Variablen heißen in Java *Klassenvariablen*, die man üblicherweise am Beginn einer Klasse (das muss aber syntaktisch nicht sein) definiert.
- Wie erwähnt, gelten diese Variablen in der gesamten Klasse (im gesamten Modul) und ggf. auch darüber hinaus, stehen also für die Bildung von Ausdrücken zur Verfügung.
- Die Definition wird wiederum von den Schlüsselwörtern **`public`** und **`static`** eingeleitet.
- Klassenvariablen kann man auch als Konstanten definieren, wie bei lokalen Konstanten dient hierzu das Schlüsselwort **`final`**

Globale Konstante – Beispiel

```
public class Kreis {
    /**
     * Die Konstante spezifiziert die Kreiszahl pi und ist
     * als Kreis.PI auch ausserhalb dieser Klasse bekannt.
     */
    public static final double PI = 3.14159;

    /**
     * Berechnung des Umfangs eines Kreises mit gegebenem Radius.
     * @param radius Der Radius des Kreises.
     * @return Der Umfang des Kreises.
     */
    public static double kreisUmfang(double radius) {
        return 2 * PI * radius;
    }
}
```

Globale Variable – Beispiel

```
public class OhneGlobaleVar {

  public static void add(int x) {
     int sum = 0;
     sum = sum + x;
  }

  public static void main(String[] args)
  {
    int sum = 0;
    add(3);
    add(5);
    add(7);
    System.out.println("Summe: "+sum);
  }
}
```

```
public class MitGlobalerVar {

  public static int sum = 0;

  public static void add(int x) {
     sum = sum + x;
  }

  public static void main(String[] args)
  {
     sum = 0;
     add(3);
     add(5);
     add(7);
     System.out.println("Summe: "+sum);
  }
}
```

Ausgabe: `Summe: 0`

Ausgabe: `Summe: 15`

Globale Größen

- Im Gegensatz zu lokalen Variablen muss man Klassenvariablen nicht explizit initialisieren.
- Sie werden dann automatisch mit ihren Standardwerten initialisiert:

Typ	Standardwert
`boolean`	`false`
`char`	`u0000`
`byte, short, int, long`	`0`
`float, double`	`0.0`

- Klassenkonstanten müssen dagegen explizit initialisiert werden.

Sichtbarkeit globaler Größen

- Lokale Variablen innerhalb einer Klasse können genauso heißen wie eine Klassenvariable.
- Beispiel:

```
public class Sichtbarkeit {
      public static int variablenname;

      public static void main(String[] args) {
            boolean variablenname = true;
      }
}
```

- Das bedeutet: Während bei lokalen Variablen Sichtbarkeit und Gültigkeit zusammenfallen, muss man zwischen beiden Eigenschaften bei Klassenvariablen prinzipiell unterscheiden.

Sichtbarkeit globaler Größen

- Das ist kein Widerspruch zum Verbot, den gleichen Namen innerhalb des Gültigkeitsbereichs einer Variable nochmal zu verwenden, denn genau genommen heißt die Klassenvariable anders.
- Zu ihrem Namen gehört der vollständige Klassenname, in dem die Variable/Konstante definiert wurde (übrigens inklusive des Package-Namens, was ein Package ist, lernen wir aber — Sie ahnen es — erst später).
- Unter dem vollständigen Namen ist eine globale Größe auch dann sichtbar, wenn der innerhalb der Klasse geltende Name durch den identisch gewählten Namen einer lokalen Variable verdeckt ist.

Sichtbarkeit globaler Größen

LMU

- I.Ü. gilt diese Namensregel analog für unsere (statischen) Methoden; dadurch können auch Methoden, die gleich benannt sind (und sogar die selbe Signatur haben) aber in unterschiedlichen Klassen vereinbart wurden, eindeutig unterschieden werden.
- Das ist nicht ganz unwichtig schließlich könnte es in einem großen Software-Kosmos dann nur jeweils eine einzige Methode mit einer speziellen Signatur `type m(inputType inputParam)` geben.
- Vorsicht, das hat nix — äh tschuldigung: nichts — mit dem Konzept des Überladens zu tun!

Beispiele

- Der (vorläufig) vollständige Name der Konstanten `PI` aus der Klasse `Kreis` ist also: `Kreis.PI` und stellt die Kreiszahl allen Klassen zur Verfügung.
- Das uns bereits bekannte Modul `Math` stellt zwei Konstanten zur Verfügung:
  - die Eulersche Zahl `Math.E`
    (The double value that is closer than any other to *e*, the base of the natural logarithms)
  - und ebenfalls die Kreiszahl `Math.PI`
    (The double value that is closer than any other to π, the ratio of the circumference of a circle to its diameter)

Namen statischer Elemente

- Beispiel mit Methoden
  Die beiden Methoden `add` der Klassen `OhneGlobaleVar` und `MitGlobalerVar` haben auf den ersten Blick exakt dieselbe Signatur, können aber anhand des (vorläufig) vollständigen Namens eindeutig unterschieden werden:

  ```
  OhneGlobaleVar.add
  MitGlobalerVar.add
  ```

- Statische Methoden und statische (globale) Variablen/Konstanten heißen auch Klassenmethoden bzw. Klassenvariablen/-konstanten.

Zustände globaler Größen

- Wie passen globale Größen in unser Zustandsmodell?
- Wir müssen nur die Menge der Namen $N(\mathcal{S})$ um die globalen Variablen und Konstanten erweitern, ansonsten werden sie behandelt wie lokale Variablen, d.h. die Semantik der Zustandsübergänge gilt analog.
- Einziger Unterschied: sie sind wirklich überall sichtbar/gültig, auch innerhalb des Rumpfes einer aufgerufenen Methode.

Zustände globaler Größen

- Sind wirklich alle globalen Variablen/Konstanten immer bekannt (d.h. die entsprechenden Zettel existieren)?
- Es würde wenig Sinn machen alle Variablen/Konstanten aller Module, die es auf der Welt gibt, in $N(\mathcal{S})$ aufzunehmen (was für eine Papierverschwendung!!!).
- Eine ähnliche Problematik ergibt sich übrigens bei den Methoden anderer Klassen.

Zustände globaler Größen

- Tatsächlich muss man globale Größen und Methoden anderer Module (Klassen) explizit *laden*, d.h. dem Compiler sagen, dass man sie verwenden möchte.
- Dies funktioniert in Java mit der Anweisung **`import`**, die wir in Zusammenhang mit Packages noch kennen lernen werden.
- Nur schon mal soviel: es gibt Module (aus der Standardbibliothek), die automatisch geladen werden (z.B. `Math`), sodass diese nicht explizit geladen werden müssen (d.h. deren Klassenvariablen/-konstanten stehen implizit zur Verfügung bzw. sind immer Teil von $N(\mathcal{S})$).

Vorsicht mit globalen Größen

- *Achtung*: Globale Variablen möglichst vermeiden, ansonsten nur, wenn
  - sie in mehreren Methoden verwendet werden müssen
  - ihr Wert zwischen Methodenaufrufen erhalten bleiben muss
- Wann immer möglich, lokale Variablen verwenden
  - Einfachere Namenswahl
  - Bessere Lesbarkeit: Deklaration und Verwendung der Variablen liegen nahe beieinander
  - Keine Nebeneffekte: Lokale Variablen können nicht durch andere Methoden versehentlich überschrieben/verändert werden