Übersicht

4.4 Imperative Algorithmen

- Die in Kapitel 4.3 informell eingeführten imperativen Konzepte werden nun etwas genauer betrachtet
- Wir verfeinern zunächst das (imperative) Konzept der Variablen und erarbeiten uns dann das Konzept der Prozeduren als Verallgemeinerung von Funktionen
- Wir verwenden dabei gleich die Syntax von Java. Wie wir (einfache) Ausdrücke in Java bilden können, haben wir ja bereits kennengelernt (Kap. 4.1.4)

Übersicht

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- Betrachten wir als Beispiel folgende Funktion

  $f(x)$ für $x \neq -1$ mit f: $\mathbb{R} \rightarrow \mathbb{R}$ gegeben durch

$$f(x) = \left(x + 1 + \frac{1}{x+1}\right)^2 \quad \text{für } x \neq -1$$

  - Eine imperative Darstellung erhält man durch Aufteilung der Funktionsdefinition in „Zwischenberechnungen“:

$$\begin{aligned} y_1 &= x + 1 \\ y_2 &= y_1 + \tfrac{1}{y_1} \\ y_3 &= y_2 * y_2 \\ f(x) &= y_3. \end{aligned}$$

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- Intuition des Auswertungsvorgangs der imperativen Darstellung:
  - y_1, y_2 *und* y_3 repräsentieren drei Zettel.
  - Auf diese Zettel werden der Reihe nach Rechenergebnisse geschrieben.
  - Bei Bedarf wird der Wert auf dem Zettel "abgelesen".
- Wir hatten schon bemerkt, dass formal hinter dieser Intuition eine Substitution steckt:

  x wird beim Aufruf der Funktion mit dem Eingabewert substituiert

  $[y_1 / (x + 1)]$, beim Aufruf ist damit der Wert von y_1 wohldefiniert

  $[y_2 / (y_1 + 1/y_1)]$, beim Aufruf ist damit der Wert von y_2 wohldefiniert

  …

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- Bei genauerer Betrachtung:
  - Nachdem der Wert von y_1 zur Berechnung von y_2 benutzt wurde, wird er im folgenden nicht mehr benötigt.
  - Eigentlich könnte der Zettel nach dem Verwenden (Ablesen) "radiert" und für die weiteren Schritte wiederverwendet werden.
  - In diesem Fall kämen wir mit einem Zettel *y* aus.

$$y = x + 1$$
$$y = y + \frac{1}{y}$$
$$y = y * y$$

- Es gibt also offenbar radierbare und nicht-radierbare Zettel
  - Radierbare Zettel heißen in diesem (imperativen) Zusammenhang *Variablen*
  - Nicht-radierbare Zettel heißen in diesem (imperativen) Zusammenhang *Konstanten*

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- Variablen und Konstanten können *deklariert* werden:

  var <*VName*> : <*Typ*>;

  Dabei wird ein "leerer Zettel" (letztlich eine freie Speicherzelle) angelegt.
  Formal: Der Bezeichner <*VName*> wird der Menge der zur Verfügung stehenden Variablen V, die wir in Ausdrücken verwenden dürfen, hinzugefügt.
- Variablen und Konstanten können durch eine erstmalige Wertzuweisung *initialisiert* werden:

  <*VName*> = <*Ausdruck*>;

  Dabei wird ein Wert des Ausdrucks <*Ausdruck*> auf den "Zettel" (letztlich in die Speicherzelle) geschrieben (<*Ausdruck*> muss vom Typ <*Typ*> sein).
  Formal: Die Variable <*VName*> wird durch den Ausdruck <*Ausdruck*> substituiert: [<*VName*> / <*Ausdruck*>]

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- Der Wert einer Variablen kann später auch noch durch eine weitere Wertzuweisung verändert werden.
  - Dabei wird der alte Wert auf dem Zettel radiert und der Wert des neuen Ausdrucks auf den Zettel geschrieben.
  - Formal also eine weitere Substitution
- Nach der Deklaration kann der Wert einer Konstanten nur noch einmalig durch eine Wertzuweisung verändert / initialisiert werden.
- Nach der Deklaration kann der Wert einer Variablen beliebig oft durch eine Wertzuweisung verändert werden.

- Im Übrigen hat jede Variable / Konstante natürlich einen Typ.
- Eine Deklaration ist also das Bereitstellen eines “Platzhalters” des entsprechenden Typs.

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- Eine *Variablendeklaration* hat in Java die Gestalt

```
Typname variablenName;
```

Konvention: Variablennamen beginnen mit kleinen Buchstaben.

- Eine *Konstantendeklaration* hat in Java die Gestalt

```
final Typname KONSTANTENNAME;
```

Konvention: Konstantennamen bestehen komplett aus großen Buchstaben.

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- Eine *Wertzuweisung* (z.B. Initialisierung) hat in Java die Gestalt

```
variablenName = NeuerWert;
```

bzw.

```
KONSTANTENNAME = Wert;
```

(nur als Initialisierung)

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

```
Typname variablenName = InitialerWert;
```

(Konstantendeklaration mit Initialisierung analog mit Zusatz **`final`**)

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- Beispiele
  - Konstanten: **final** <typ> <NAME> = <ausdruck>;

```
final double K_1 = 1;
final double K_2 = K_1 + 1 / K_1;
final double K_3 = K_2 * K_2;
final double BESTEHENSGRENZE_PROZENT = 0.5;
final int GESAMTPUNKTZAHL;
GESAMTPUNKTZAHL = 80;
```

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

```
double y;
y = 1.0;
int klausurPunkte = 42;
boolean klausurBestanden =
        ( ((double) klausurPunkte) /
          ((double) GESAMTPUNKTZAHL) )
                >= BESTEHENSGRENZE_PROZENT;
```

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- Für bestimmte einfache Operationen (Addition und Subtraktion mit 1 als zweitem Operanden) gibt es gängige 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`

- Will man eine Variable nicht nur um 1 erhöhen/verringern, sondern allgemein einen neuen Wert zuweisen, der vom alten Wert abhängig ist, gibt es Kurznotationen wie folgt:

Operator	Bezeichnung	Bedeutung
`+=`	Summe	`a+=b` weist `a` die Summe von `a` und `b` zu
`-=`	Differenz	`a-=b` weist `a` die Differenz von `a` und `b` zu
`*=`	Produkt	`a*=b` weist `a` das Produkt aus `a` und `b` zu
`/=`	Quotient	`a/=b` weist `a` den Quotienten von `a` und `b` zu
(Auch für weitere Operatoren möglich...)		

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- Beipiel

```
int x = 3;
int y;

x--;                // entspricht x = x-1;
                    // Wert des Ausdrucks ist 3,
                    // x hat anschliessend den Wert 2

y=x++;              // Wert des Ausdrucks x++ ist 2
                    // d.h. y wird mit 2 initialisiert
                    // x hat anschliessend den Wert 3

x+=y;               // x hat anschliessend den Wert 5
```

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- Wie wir bereits festgestellt haben, steht eine Anweisung für einen einzelnen Abarbeitungsschritt in einem Algorithmus.
- Java kombiniert imperative und funktionale Konzepte, folgt aber einer imperativen Auffassung von Programmen.
- Wir haben schon einige Arten von Anweisungen kennengelernt, die es auch in Java gibt:
  - Die leere Anweisung
  - Vereinbarungen und Initialisierungen von Variablen/Konstanten
  - 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. Ausdrücke mit Nebeneffekten sind:
    - Wertzuweisung sowie (Prä- / Post-)Inkrement und Dekrement
    - Funktions-/Prozeduraufruf (werden wir später kennenlernen),
    - Instanzerzeugung (werden wir später kennenlernen)

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- Ein *Block* von Anweisungen wird in Java gebildet von einer öffnenden geschweiften Klammer und einer schließenden geschweiften Klammer, die eine beliebige Menge von Anweisungen umschließen:

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

- 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 (z.B. in einer bedingten Anweisung).
- Eine Anweisung in einem Block kann natürlich auch wieder ein Block sein, d.h. Blöcke können ineinander verschachtelt werden.

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- 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* (Zettel) zu Verfügung stellt.
- Die *Gültigkeit* einer Variablen erstreckt sich auf alle Programmstellen, an denen der Name der Variablen dem Compiler durch eine Vereinbarung (Deklaration) bekannt ist. An diesen Programmstellen dürfen Ausdrücke mit dieser Variablen gebildet werden
- Die *Sichtbarkeit* einer Variablen erstreckt sich auf alle Programmstellen, an denen man über den Namen der Variablen auf den durch sie benannten Ausdruck (und damit auf dessen Wert) zugreifen kann.

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- Eine in einem Block deklarierte Variable ist ab ihrer Deklaration bis zum Ende des Blocks *gültig* und *sichtbar*. Daher heißt diese Variable auch *lokale* Variable.
- Mit Verlassen des Blocks, in dem eine lokale Variable 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 und für neue Variablen verwendet werden kann.
- Solange eine lokale Variable sichtbar ist, darf keine neue lokale Variable gleichen Namens angelegt werden.

4.4 Imperative Algorithmen

4.4.1 Variablen und Konstanten

- Beispiel

```
...
int i = 0;
{
    int i = 1; // nicht erlaubt
    i = 1;     // erlaubt
    int j = 0;
}
j = 1;     // nicht moeglich
...
```

Übersicht

4.4 Imperative Algorithmen

4.4.2 Prozeduren

- Die Prozedur als Verallgemeinerung der Funktion dient zur *Abstraktion* von Algorithmen (oder von einzelnen Schritten eines Algorithmus).
- Durch Parametrisierung wird von der Identität der Daten abstrahiert: die Berechnungsvorschriften werden mit abstrakten (Eingabe-) Parametern formuliert, konkrete Eingabedaten bilden die aktuellen (Parameter-) Werte mit denen die Eingabedaten dann beim Aufruf substituiert werden.
- Durch Spezifikation des (Ein- / Ausgabe-) Verhaltens wird von den Implementierungsdetails abstrahiert: Vorteile sind
  - Örtliche Eingrenzung (*Locality*): Die Implementierung einer Abstraktion kann verstanden oder geschrieben werden, ohne die Implementierungen anderer Abstraktionen kennen zu müssen.
  - Änderbarkeit (*Modifiability*): Jede Abstraktion kann reimplementiert werden, ohne dass andere Abstraktionen geändert werden müssen.
  - Wiederverwendbarkeit (*Reusability*): Die Implementierung einer Abstraktion kann beliebig wiederverwendet werden.

4.4 Imperative Algorithmen

4.4.2 Prozeduren

- Wie bereits erwähnt ist eine Funktion eine Prozedur ohne Seiteneffekte, d.h. die Prozedur ist das allgemeinere Konzept.
- Wir haben bereits Prozeduren mit Seiteneffekten kennengelernt:

```
public class HelloWorld
{
  public static final String GRUSS = "Hello, World!";

  public static void main(String[] args)
  {
    System.out.println(GRUSS);
  }
}
```

- Die Prozedur `main` hat als Seiteneffekt die Ausgabe von „`Hello World!`“ auf der Kommandozeile.

4.4 Imperative Algorithmen

4.4.2 Prozeduren

- Bei einer Funktion ist der Bildbereich eine wichtige Information:

$$f : D \rightarrow B$$

- Bei einer Prozedur, die keine Funktion ist, wählt man als Bildbereich oft die leere Menge:

$$p : D \rightarrow \emptyset$$

- Dies signalisiert, dass die Seiteneffekte der Prozedur zur eigentlichen Umsetzung eines Algorithmus gehören, dagegen aber kein (bestimmtes) Element aus dem Bildbereich einer Abbildung als Ergebnis des Algorithmus angesehen werden kann.
- Sehr häufig findet man in imperativen Implementierungen von Algorithmen aber eine Mischform, in der eine Prozedur sowohl Seiteneffekte hat als auch einen nicht-leeren Bildbereich.
- Manchmal wird auch $\mathbb{B}$ als Ergebnistyp gewählt. Man zeigt dann mit dem Ergebniswert, ob der beabsichtigte Nebeneffekt erfolgreich war.

4.4 Imperative Algorithmen

4.4.2 Prozeduren

- In Java werden Prozeduren (und damit auch Funktionen) durch *Methoden* realisiert.
- Eine Methode wird definiert durch
  - den *Methodenkopf*:

    `public static` `<typ> <name>(<parameterliste>)`

    der den Namen und die *Signatur* der Methode spezifiziert:

    `<typ>` ist der Bildbereich (Ergebnistyp)

    `<parameterliste>` spezifiziert die Eingabeparameter und –typen und besteht aus endlich vielen (auch keinen) Paaren von Typen und Variablennamen (`<Typ> <VName>`) jeweils durch Komma getrennt
  - den *Methodenrumpf*: einen Block von Anweisungen (also in `{ }` eingeschlossen), der sich an den Methodenkopf anschließt.
- Beispiel:

```
public static int mitte(int x, int y, int z) { … }
```

4.4 Imperative Algorithmen

4.4.2 Prozeduren

- Als besonderer Ergebnistyp einer Methode ist **`void`** (für ∅) möglich, d.h. eine Methode mit Ergebnistyp **`void`** gibt *kein* Ergebnis zurück. Hier liegt der Sinn 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>` ≠ **`void`** hat, muss mind. eine **`return`**-Anweisung mit einem Ausdruck vom Typ `<type>` haben

- Bemerkung: Sie wundern sich vielleicht, dass wir die funktionalen Konzepte nicht anhand der „Java-Syntax“ eingeführt haben, die imperativen Konzepte nun aber schon. Einer der Hauptgründe dafür ist, dass es ***in Java offenbar keine eigene („reine“) Realisierung des Konzepts der Funktion***.

4.4 Imperative Algorithmen

4.4.2 Prozeduren

- Beispiel:

```
/**
 * Methode zur Berechnung der zurueckgelegten Strecke
 * nach Einwirken der Kraft <code>k</code>
 * auf einen Koerper der Masse <code>m</code>
 * fuer die Zeitdauer <code>t</code>.
 *
 * @param m Masse des Koerpers
 * @param t Zeitdauer
 * @param k Kraft
 * @return zurueckgelegte Strecke
 */
public static double strecke(double m, double t, double k)
{
  double b = k / m;
  return 0.5 * b * (t * t);
}
```

Methodenkopf

Methodenrumpf

4.4 Imperative Algorithmen

4.4.2 Prozeduren

- Beispiel

```
{
  /**
   * Methode zur Berechnung der Funktion
   * <i>f(x) = ((x + 1) + 1 / (x + 1))<sup>2</sup></i>.
   *
   * @param x der Eingabewert
   * @return Wert der Funktion f an der Stelle x
   */
  public static double f(double x)
  {
    double y = x + 1;
    y = y + 1/y;
    y = y * y;
    return y;
  }
```

4.4 Imperative Algorithmen

4.4.2 Prozeduren

- Eine Methode kann in Java wiederum aufgerufen werden. Solch ein *Methodenaufruf* ist syntaxtisch ein Ausdruck, kann also überall stehen, wo ein Ausdruck stehen darf (die formale Erweiterung der induktiven Definition von Ausdrücken ersparen wir uns):
  - Eine Methode mit Ergebnistyp `<typ>` ≠ **`void`** ist ein Ausdruck vom Typ `<typ>` und kann überall dort stehen, wo ein Ausdruck vom Typ `<typ>` verlangt ist (z.B. eine Wertzuweisung an eine Variable vom Typ `<typ>`)

    Beispiel: **`double`** `s = strecke(3.0, 4.2, 7.1);`

    Dies entspricht einem Funktionsaufruf, kann jetzt aber Seiteneffekte haben!
  - Eine Methode mit Ergebnistyp **`void`** ist ein Ausdruck vom „Typ“ **`void`** und kann als Ausdrucksanweisung verwendet werden. Die Seiteneffekte der Methode ist das beabsichtigte Resultat der Anweisung.

    Beispiel: `System.out.println` ist eine Methode mit Ergebnistyp **`void`** und hat als Seiteneffekt den Eingabeparameter (Typ `String` für Zeichenketten) auf dem Bildschirm auszugeben, d.h.

    `System.out.println(…);` ist eine Ausdrucksanweisung

4.4 Imperative Algorithmen

4.4.2 Prozeduren

- Genauso wie bei Funktionen stellt sich auch für die Prozeduren die Frage: was passiert bei einem *Prozeduraufruf*?
- Wir klären diese Frage zunächst wieder informell mit unserer Intuition, dass eine Variable/Konstante ein Zettel ist
- Offenbar bewirkt eine Anweisung eine *Veränderung des Zustands* indem sich ein Programm (in Ausführung) gerade befindet
- Eine Wertzuweisung an eine Variable z.B. verändert den Zustand so, dass die Variable einen neuen Wert bekommt (also ab jetzt neu substituiert wird).
- Stellt sich noch die Frage, wie so ein Zustand aussehen kann.

4.4 Imperative Algorithmen

4.4.2 Prozeduren

- Formal ist ein *Zustand* eine Menge **Z** von Paaren $z = (x, d)$ mit folgenden Eigenschaften
  - x ist ein Eingabeparameter, eine (vereinbarte) Konstante oder eine (vereinbarte) Variable und heißt *Name* von z
  - d ist ein Objekt der Sorte von x (oder undefiniert, notiert als „ω“) und heißt *Inhalt* von z
  - Die Menge Z enthält keine zwei verschiedenen Paare (x, d_1) und (x, d_2) mit gleichem Namen x
- Ein Paar (x, d) formalisiert das Bild eines Zettels x auf dem der Wert d steht.
- (x, ω) kann als „leerer Zettel“ verstanden werden

4.4 Imperative Algorithmen

4.4.2 Prozeduren

- Sei $N(\mathbf{Z})$ die Menge aller Namen (Variablen und Konstanten) in $\mathbf{Z}$.
- Ein Zustand $\mathbf{Z}$ definiert eine Substitution für alle Namen $x \in N(\mathbf{Z})$ in $\mathbf{Z}$ deren Wert $d \neq \omega$ ist.
- Dies kommt Ihnen bekannt vor:
  - Bei Funktionen in Kap. 4.2 haben wir gesehen, dass der Aufruf eines Algorithmus eine Substitution σ für die Eingabeparameter des Algorithmus $V(\sigma)$ spezifiziert (bzw. jetzt für $V(\mathbf{Z})$).
  - Zustände beinhaltet offenbar nun aber nicht mehr nur die Eingabeparameter sondern auch die vereinbarten Konstanten und Variablen, daher ist $N(\mathbf{Z})$ eine Obermenge von $V(\sigma)$ bzw. $V(\mathbf{Z})$, d.h. $V(\mathbf{Z}) \subseteq N(\mathbf{Z})$.
  - Daher benutzen wir unterschiedliche Symbole σ und $\mathbf{Z}$ bzw. N und V.
- Offensichtlich ist $(e, d) \in \mathbf{Z}$ mit $d \neq \omega$ für alle *Eingabeparameter* $e \in V(\mathbf{Z})$ aber nicht notwendigerweise für alle vereinbarten *lokalen Variablen und Konstanten* $N(\mathbf{Z}) \setminus V(\mathbf{Z})$.