Überblick

Road Map

- Wir wenden uns jetzt dem imperativen Paradigma zu
- Zur Erinnerung: imperative Algorithmen werden typischerweise als Folge von *Anweisungen* formuliert
- Diese Anweisung haben meist *Nebeneffekte*, bei denen Größen geändert werden
- Dabei handelt es sich meist um *Variablen*
- Wir werden im Übrigen sehen, dass einiger der funktionalen Konzept der vergangenen Kapitel entsprechende imperative Pendants haben, bzw. dass sich funktionale und imperative Konzepte sehr gut miteinander vereinbaren lassen

Variablen

- Im vorherigen Kapitel haben wir Ausdrücke (in Java) nur mit Operationssymbolen (Literale und mehrstelligen Operatoren) gebildet; als Variablen haben wir nur die Eingabevariablen der Algorithmen (Funktionen) zugelassen
- Diese Einschränkung über der Menge der für Ausdrücke zur Verfügung stehenden Variablen V geben wir jetzt auf
- Wir erlauben nun auch weitere Variablen in Ausdrücken, allerdings müssen diese vorher *vereinbart*, also bekannt, sein

Variablen

- Wozu sind Variablen gut?
- Betrachten wir als Beispiel folgenden (in funktionaler Darstellung) angegebenen Algorithmus:
  - Berechne die Funktion $f(x)$ für $x \neq -1$ mit $f : \mathbb{R} \to \mathbb{R}$ gegeben durch

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

  - Eine imperative Darstellung erhält man z.B. durch Aufteilung der Funktionsdefinition in mehrere *Anweisungen*, die nacheinander auszuführen sind:

$$y_1 = x + 1$$
$$y_2 = y_1 + \frac{1}{y_1}$$
$$y_3 = y_2 * y_2$$
$$f(x) = y_3.$$

Variablen

- 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
- Formal steckt hinter dieser Intuition eine Substitution
  - x wird beim Aufruf der Funktion wie bisher durch den Eingabewert substituiert
  - y_1 wird mit dem Wert des Ausdrucks $x + 1$ substituiert wobei x bereits substituiert wurde (der Wert von y_1 ist damit beim Aufruf von f wohldefiniert)
  - Mit y_2 und y_3 kann man analog verfahren

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:

$$y = x + 1$$
$$y = y + \frac{1}{y}$$
$$y = y * y$$
$$f(x) = y.$$

- Es gibt also offenbar *radierbare* und *nicht-radierbare* Zettel
- Radierbare Zettel heißen *Variablen*
- Nicht-radierbare Zettel heißen *Konstanten*

Variablen und Konstanten

- Variablen und Konstanten können *deklariert* werden
  - Intuitiv wird ein *leerer Zettel* (Speicherzelle) angelegt
  - Formal bedeutet die Deklaration einer Variablen/Konstanten v, dass der Bezeichner v zu der Menge der zur Verfügung stehenden Variablen V, die wir in Ausdrücken verwenden dürfen, hinzugefügt wird
  - Natürlich hat jede Variable einen Typ, daher fordern wir, dass der Typ bei der Deklaration angegeben werden muss
- Variablen und Konstanten können *intialisiert* werden
  - Intuitiv wird ein Wert auf den Zettel (in die Speicherzelle) geschrieben
  - Dadurch wird der Variablen/Konstanten $v \in V$ der Wert eines Ausdrucks a zugewiesen (dies nennt man auch *Wertzuweisung*)
  - Formal bedeutet dies, dass v durch a substituiert wird, d.h. $[v/a]$

Variablen und Konstanten

- Der Wert einer Variablen kann später auch noch durch eine (neue) *Wertzuweisung* verändert werden
  - Intuitiv wird der alte Wert auf dem Zettel radiert und der Wert des neuen Ausdrucks auf den Zettel geschrieben
  - Formal also eine erneute 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.

Variablen und Konstanten in Java

- Eine Variablendeklaration hat in Java die Gestalt

```
Typname variablenName;
```

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

- Eine Konstantendeklaration hat in Java die Gestalt

```
final Typname KONSTANTENNAME;
```

  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

- Eine Wertzuweisung (z.B. Initialisierung) hat 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`**)

- Generell sind Deklaration, Initialisierung und Wertzuweisung *Anweisungen* (daher werden sie in Java mit einem Semikolon beendet)

Variablen und Konstanten in Java

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;
```

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

```
double y = x + 1; //Achtung: was ist x???
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`

Ahhh, endlich machen diese Operationen Sinn . . .

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	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

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

Anweisungen

- 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
- Einige wichtige Arten von Anweisungen in Java sind
  - Die *leere* Anweisung, bestehend aus einem einzigen `;`
  - 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, z.B. Wertzuweisung sowie (Prä- / Post-)Inkrement und Dekrement von Variablen/Konstanten, Funktions-/Prozeduraufruf (werden wir später kennenlernen), Instanzerzeugung (werden wir später kennenlernen)

Der Anweisungsblock

- 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
- Blöcke können beliebig geschachtelt sein, d.h. eine Anweisung in einem Block kann auch wieder ein Block sein

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 Vereinbarung (*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.

Gültigkeitsbereich von Variablen

- Beispiel:

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

Überblick

Road Map

- Wir kennen nun das Konzept der Variablen/Konstanten als Container (Zettel) für zu verändernde Größen
- Wir wissen, dass die Veränderung der Werte von Variablen durch (einfache) Anweisungen (Deklaration, Initialisierung u.v.a. Wertzuweisung) ermöglicht wird
- Wir erweitern nun den Funktionsbegriff um die Idee der imperativen Algorithmen zu formalisieren

Prozeduren

- Die *Prozedur* als Verallgemeinerung der Funktion dient zur *Abstraktion* von Algorithmen (oder von einzelnen Schritten eines Algorithmus).
- Wie bei Funktionen wird durch Parametrisierung von der Identität der Daten abstrahiert
  - die Berechnungsvorschriften werden mit abstrakten (Eingabe-) Parametern (Variablen) formuliert, so wie bei Funktionen der Rumpf mit den (Eingabe-) Parametern gebildet wurde
  - 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

Prozeduren

Vorteile der Abstraktion durch Prozeduren:

- 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.

Prozeduren

- Funktionen und Prozeduren haben also gewisse Gemeinsamkeiten, aber: eine Funktion kann man als Prozedur bezeichnen, nicht jede Prozedur ist jedoch eine Funktion.
- Eine Funktion stellt nur eine Abbildung von Elementen aus dem Definitionsbereich auf Elemente aus dem Bildbereich dar.
- Es werden aber keine Werte verändert.
- Im imperativen Paradigma können Werte von Variablen verändert werden (durch Anweisungen). Dies kann Effekte auf andere Bereiche eines Programmes haben.
- Treten in einer Prozedur solche sog. Seiteneffekte (oder Nebeneffekte) auf, kann man nicht mehr von einer Funktion sprechen.
- Eine Funktion kann man also als eine Prozedur ohne Seiteneffekte auffassen.

Funktionen und Prozeduren

- Wir haben bereits Prozeduren mit Seiteneffekten verwendet:

```
public class HelloWorld
{

  public static void main(String[] args)
  {
    System.out.println("Hello, World!");
  }
}
```

- Die Methode `main` in diesem Beispiel ist eine Prozedur und hat als Seiteneffekt die Ausgabe von `"Hello World!"` auf der Kommandozeile.

Funktionen und Prozeduren

- Bei einer Funktion ist der Bildbereich eine wichtige Information:

$$f : D \to B$$

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

$$p : D \to \emptyset$$

  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 als Ergebnistyp $\mathbb{B}$ gewählt, das Ergebnis soll dann anzeigen, ob der beabsichtigte Nebeneffekt erfolgreich war

Prozeduren in Java

- In Java werden Prozeduren (wie 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 (in entspr. Klammern), der sich an den Methodenkopf anschließt.
- Beispiel: `public static int mitte(int x, int y, int z) ...`

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 um Prozeduren erweitern
- Dies ermöglicht formal, benutzereigene Prozeduren (imperative Algorithmen) über Module bereitzustellen
- Ein (zugegeben eher sinnfreies) Beispiel:

```
public class PhysikImperativ
{
  /**
   * ...
   */
  public static double streckeImperativ(double m, double t, double k)
  {
    double b = k / m;
    return 0.5 * b * (t * t);
  }

  ...
}
```