Teil I

Konzepte imperativer Programmierung

Abschnitt 4: Imperative Programmierung

Vorbemerkung

- Induktion, Rekursion und das Konzept der (induktiv definierten) Ausdrücke sind funktionale Konzepte und werden als solche in der Einführungsvorlesung “Programmierung und Modellierung” vertieft.
- Allerdings spielen diese (und andere funktionale) Konzepte auch im imperativen Programmierparadigma eine wichtige Rolle.
- Auch als Entwurfsmuster spielt Rekursion in imperativen Sprachen eine wichtige Rolle.
- Wir wenden uns nun aber dem imperativen Programmierparadigma und der Realisierung am Beispiel von Java zu.

Überblick

Grunddatentypen

- Typischerweise stellt jede höhere Programmiersprache gewisse *Grunddatentypen* zur Verfügung.
- Rein konzeptionell handelt es sich dabei um eine Menge von Sorten.
- Zusätzlich zu diesen Grunddatentypen werden auch gewisse *Grundoperationen* bereitgestellt, also eine Menge von (teilweise überladenen) Operationssymbolen, die auf die bereitgestellten Datentypen (Sorten) angewandt werden können.
- Die Semantik dieser Operationen ist durch den zugrundeliegenden Algorithmus zur Berechnung der entsprechenden Funktion definiert.
- Aus den Literalen der Grunddatentypen und den zugehörigen Grundoperationen können nun, wie im vorherigen Abschnitt definiert, Ausdrücke gebildet werden.
- Achtung: Diese Ausdrücke enthalten zunächst noch keine Variablen. Zu Variablen kommen wir später.

Grunddatentypen (primitive Typen) in Java

- Wir wissen bereits, dass es in Java Grunddatentypen (auch *atomare* oder *primitive Typen*) für $\mathbb{B}$, *CHAR*, eine Teilmenge von $\mathbb{Z}$ und für eine Teilmenge von $\mathbb{R}$ gibt, aber keinen eigenen Grunddatentyp für $\mathbb{N}$.
- Die Werte der einzelnen primitiven Typen werden intern binär dargestellt.
- Die Datentypen unterscheiden sich u.a. in der Anzahl der Bits, die für ihre Darstellung verwendet werden.
- Die Anzahl der Bits, die für die Darstellung der Werte eines primitiven Datentyps verwendet werden, heißt *Länge* des Typs.
- Die Länge eines Datentyps hat Einfluss auf den Wertebereich des Typs.
- Als *objektorientierte* Sprache bietet Java zusätzlich die Möglichkeit, benutzerdefinierte (sog. *abstrakte*) Datentypen zu definieren. Diese Möglichkeiten lernen wir im Teil über objektorientierte Modellierung genauer kennen.

Grunddatentypen (primitive Typen) in Java

Überblick

Typname	Länge	Wertebereich
`boolean`	1 Byte	Wahrheitswerte {`true`,`false`}
`char`	2 Byte	Alle Unicode-Zeichen
`byte`	1 Byte	Ganze Zahlen von -2^7 bis $2^7 - 1$
`short`	2 Byte	Ganze Zahlen von -2^{15} bis $2^{15} - 1$
`int`	4 Byte	Ganze Zahlen von -2^{31} bis $2^{31} - 1$
`long`	8 Byte	Ganze Zahlen von -2^{63} bis $2^{63} - 1$
`float`	4 Byte	Gleitkommazahlen (einfache Genauigkeit)
`double`	8 Byte	Gleitkommazahlen (doppelte Genauigkeit)

Wahrheitswerte in Java

- Java hat einen eigenen Typ `boolean` für Wahrheitswerte.
- Die beiden einzigen Werte (*Konstanten*) sind `true` für "wahr" und `false` für "falsch".

Boole'sche Operatoren in Java

Operator	Bezeichnung	Bedeutung
`!`	logisches NICHT (¬)	`!a` ergibt **`false`** wenn `a` wahr ist, sonst **`true`**.
`&`	logisches UND (∧)	`a & b` ergibt **`true`**, wenn sowohl `a` als auch `b` wahr ist. Beide Teilausdrücke (`a` und `b`) werden ausgewertet.
`&&`	sequentielles UND	`a && b` ergibt **`true`**, wenn sowohl `a` als auch `b` wahr ist. Ist `a` bereits falsch, wird **`false`** zurückgegeben und `b` nicht mehr ausgewertet.
`|`	logisches ODER (∨)	`a | b` ergibt **`true`**, wenn mindestens einer der beiden Ausdrücke `a` oder `b` wahr ist. Beide Teilausdrücke (`a` und `b`) werden ausgewertet.
`||`	sequentielles ODER	`a || b` ergibt **`true`**, wenn mindestens einer der beiden Ausdrücke `a` oder `b` wahr ist. Ist bereits `a` wahr, so wird **`true`** zurückgegeben und `b` nicht mehr ausgewertet.
`^`	exklusives ODER (XOR)	`a ^ b` ergibt **`true`**, wenn beide Ausdrücke `a` und `b` einen unterschiedlichen Wahrheitswert haben.

Zeichen (Character) in Java

- Java hat einen eigenen Typ **`char`** für (Unicode-)Zeichen.
- Werte (*Konstanten*) werden in einfachen Hochkommata gesetzt, z.B. `'A'` für das Zeichen "A".
- Einige Sonderzeichen können mit Hilfe von *Standard-Escape-Sequenzen* dargestellt werden:

Sequenz	Bedeutung
`\b`	Backspace (Rückschritt)
`\t`	Tabulator (horizontal)
`\n`	Newline (Zeilenumbruch)
`\f`	Seitenumbruch (Formfeed)
`\r`	Wagenrücklauf (Carriage return)
`\"`	doppeltes Anführungszeichen
`\'`	einfaches Anführungszeichen
`\\`	Backslash

Ganze Zahlen in Java

- Java hat vier Datentypen für ganze (vorzeichenbehaftete) Zahlen: **`byte`** (Länge: 8 Bit), **`short`** (Länge: 16 Bit), **`int`** (Länge: 32 Bit) und **`long`** (Länge: 64 Bit).
- Werte (*Konstanten*) können geschrieben werden in
  - Dezimalform: bestehen aus den Ziffern `0`, . . ., `9`
  - Oktalform: beginnen mit dem Präfix `0` und bestehen aus Ziffern `0`, . . ., `7`
  - Hexadezimalform: beginnen mit dem Präfix `0x` und bestehen aus Ziffern `0`, . . ., `9` und den Buchstaben `a`, . . ., `f` (bzw. `A`, . . ., `F`)
- Negative Zahlen erhalten ein vorangestelltes `-`.
- Gehört dieses `-` zum Literal?

Vorzeichen-Operatoren in Java

- Vorzeichen-Operatoren haben die Signatur

$$S \to S$$

  mit $S \in \{$**`byte`**, **`short`**, **`int`**, . . . $\}$.
- Operationen:

Operator	Bezeichnung	Bedeutung
`+`	Positives Vorzeichen	`+n` ist gleichbedeutend mit `n`
`-`	Negatives Vorzeichen	`-n` kehrt das Vorzeichen von `n` um

- Bemerkung: Offenbar sind die Operatoren überladen!!!

Gleitkommazahlen in Java

- Java hat zwei Datentypen für Fließkommazahlen: **`float`** (Länge: 32 Bit) und **`double`** (Länge: 64 Bit).
- Werte (*Konstanten*) werden immer in Dezimalnotation geschrieben und bestehen maximal aus
  - Vorkommateil
  - Dezimalpunkt (*)
  - Nachkommateil
  - Exponent `e` oder `E` (Präfix `-` möglich) (*)
    Gehört dieses `-` zum Literal?
  - Suffix `f` oder `F` (**`float`**) oder `d` oder `D` (**`double`**) (*)

  wobei mindestens einer der mit (*) gekennzeichneten Bestandteile vorhanden sein muss.
- Negative Zahlen erhalten ein vorangestelltes `-`.
- Beispiele:
  - double: `6.23`, `623E-2`, `62.3e-1`
  - float: `6.23f`, `623E-2F`, `62.2e-1f`

Arithmetische Operationen in Java

- Arithmetische Operationen haben die Signatur

$$S \times S \to S$$

  mit $S \in \{$**`byte`**, **`short`**, **`int`**, $\ldots\}$.
- Operationen:

Operator	Bezeichnung	Bedeutung
`+`	Summe	`a+b` ergibt die Summe von `a` und `b`
`-`	Differenz	`a-b` ergibt die Differenz von `a` und `b`
`*`	Produkt	`a*b` ergibt das Produkt aus `a` und `b`
`/`	Quotient	`a/b` ergibt den Quotienten von `a` und `b`
`%`	Modulo	`a%b` ergibt den Rest der ganzzahligen Division von a durch b

- Bemerkung: Offenbar sind die Operatoren überladen!!!

Vergleichsoperationen in Java

- Vergleichsoperatoren haben die Signatur

$$S \times S \rightarrow \texttt{boolean}$$

mit $S \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \ldots\}$.

- Operationen:

Operator	Bezeichnung	Bedeutung
`==`	Gleich	`a==b` ergibt **`true`**, wenn `a` gleich `b` ist
`!=`	Ungleich	`a!=b` ergibt **`true`**, wenn `a` ungleich `b` ist
`<`	Kleiner	`a<b` ergibt **`true`**, wenn `a` kleiner `b` ist
`<=`	Kleiner gleich	`a<=b` ergibt **`true`**, wenn `a` kleiner oder gleich `b` ist
`>`	Größer	`a>b` ergibt **`true`**, wenn `a` größer `b` ist
`>=`	Größer gleich	`a>=b` ergibt **`true`**, wenn `a` größer oder gleich `b` ist

- Bemerkung: Offenbar sind die Operatoren ebenfalls überladen!!!

Ausdrücke in Java

- Wir können auch in Java aus Operatoren Ausdrücke (zunächst ohne Variablen) bilden, so wie im vorherigen Kapitel besprochen.
- Laut induktiver Definition von Ausdrücken ist eine Konstante ein Ausdruck.
- Interessanter ist, aus Konstanten (z.B. den **`int`** Werten `6` und `8`) und mehrstelligen Operatoren (z.B. `+`, `*`, `<`, `&&`) Ausdrücke zu bilden. Ein gültiger Ausdruck hat selbst wieder einen Wert (der über die Semantik der beteiligten Operationen definiert ist) und einen Typ (der durch die Ergebnissorte des angewendeten Operators definiert ist):
  - `6 + 8 //Wert: 14 vom Typ int`
  - `6 * 8 //Wert: 48 vom Typ int`
  - `6 < 8 //Wert: true vom Typ boolean`
  - `6 && 8 //ungültiger Ausdruck`

  Warum?

Typkonversion

- Motivation:
  - Was passiert eigentlich, wenn man verschiedene Sorten/Typen miteinander verarbeiten will?
  - Ist der Java-Ausdruck `6 + 7.3` erlaubt?
  - Eigentlich nicht: Die Operation $+$ ist zwar überladen, d.h.

    $$+ : S \times S \to S$$

    ist definiert für beliebige primitive Datentypen S, aber es gibt keine Operation

    $$+ : S \times T \to U$$

    für *verschiedene* primitive Datentypen $S \neq T$.
- Trotzdem ist der Ausdruck `6 + 7.3` in Java erlaubt.
- *Wann* passiert *was* und *wie* bei der Auswertung des Ausdrucks `6 + 7.3`?

Automatische Typkonversion in Java

- Wann:
  Während des Übersetzens des Programmcodes durch den Compiler.
- Was:
  Der Compiler kann dem Ausdruck keinen Typ (und damit auch keinen Wert) zuweisen. Solange kein *Informationsverlust* auftritt, versucht der Compiler diese Situation zu retten.
- Wie:
  Der Compiler konvertiert automatisch den Ausdruck `6` vom Typ **`int`** in einen Ausdruck vom Typ **`double`**, so dass die Operation

  $$+ : \texttt{double} \times \texttt{double} \to \texttt{double}$$

  angewendet werden kann.

Automatische Typkonversion in Java

- Formal gesehen ist diese Konvertierung eine Operation `i->d` mit der Signatur

  $$\texttt{i->d} : \textbf{int} \rightarrow \textbf{double}$$

  d.h. der Compiler wandelt den Ausdruck
  `6 + 7.3`
  um in den Ausdruck
  `i->d(6) + 7.3`
- Dieser Ausdruck hat offensichtlich einen eindeutigen Typ und damit auch einen eindeutig definierten Wert.

Typkonversion in Java

- Was bedeuted “Informationsverlust”?
- Es gilt folgende “Kleiner-Beziehung” zwischen Datentypen:

  `byte < short < int < long < float < double`

- Beispiele:
  - `1 + 1.7` ist vom Typ **`double`**
  - `1.0f + 2` ist vom Typ **`float`**
  - `1.0f + 2.0` ist vom Typ **`double`**
- Java konvertiert Ausdrücke automatisch in den allgemeineren (“größeren”) Typ, da dabei kein Informationsverlust auftritt.
  Warum?

Explizite Typkonversion in Java: Type-Casting

- Will man eine Typkonversion zum spezielleren Typ durchführen, so muss man dies in Java explizit angeben.
- Dies nennt man allgemein *Type-Casting*.
- In Java erzwingt man die Typkonversion zum spezielleren Typ `type` durch Voranstellen von `(type)`.
- Der Ausdruck
  `(type) a`
  wandelt den Ausdruck `a` in einen Ausdruck vom Typ `type` um.
- Beispiele:
  - `(`**`byte`**`) 3` ist vom Typ **`byte`**
  - `(`**`int`**`) (2.0 + 5.0)` ist vom Typ **`int`**
  - `(`**`float`**`) 1.3e-7` ist vom Typ **`float`**

Explizite Typkonversion in Java: Type-Casting

- Bei der Typkonversion in einen spezielleren Typ kann Information verloren gehen.
- Beispiele:
  - `(`**`int`**`) 5.2` ergibt `5`
  - `(`**`int`**`) -5.2` ergibt `-5`

Der Type-Cast-Operator in Java

Im Ausdruck
`(type) a`
ist `(type)` ein Operator. Type-Cast-Operatoren bilden zusammen mit einem Ausdruck wieder einen Ausdruck.
Der Typ des Operators ist z.B.:

`(int)` : `charUbyteUshortUintUlongUfloatUdouble` → `int`

`(float)` : `charUbyteUshortUintUlongUfloatUdouble` → `float`

Sie können also z.B. auch **`char`** in **`int`** umwandeln.

Klingt komisch? Ist aber so! Und was passiert da?

Das Konzept Typkonversion im Allgemeinen

- Ganz allgemein nennt man das Konzept der Umwandlung eines Ausdrucks mit einem bestimmten Typ in einen Ausdruck mit einem anderen Typ *Typkonversion*.
- In vielen Programmiersprachen gibt es eine automatische Typkonversion meist vom spezielleren in den allgemeineren Typ.
- Eine Typkonversion vom allgemeineren in den spezielleren Typ muss (wenn erlaubt) sinnvollerweise immer explizit durch einen *Typecasting*-Operator herbeigeführt werden.
- Es gibt aber auch Programmiersprachen, in denen man grundsätzlich zur Typkonversion ein entsprechendes Typecasting explizit durchführen muss.
- Unabhängig davon kennen Sie jetzt das allgemeine Konzept und die Problematik solch einer Typkonversion. Unterschätzen Sie diese nicht als Fehlerquelle bei der Entwicklung von Algorithmen bzw. der Erstellung von Programmen!

Überblick

Ausdrücke mit Variablen

- Im vorherigen Kapitel haben wir Ausdrücke (in Java) nur mit Operationssymbolen (Konstanten und mehrstelligen Operatoren) gebildet.
- Warum haben wir dann die Ausdrücke vorher induktiv so definiert, dass darin auch Variablen vorkommen können?
- Antwort: Weil wir natürlich auch Variablen in (Java-)Ausdrücken zulassen wollen.

Konstanten und 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 durch Aufteilung der Funktionsdefinition in:

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

Konstanten und 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 durch den Eingabewert substituiert.
  - y_1 wird mit dem Wert des Ausdrucks $x + 1$ substituiert wobei x bereits substituiert wurde.
  - Mit y_2 und y_3 kann man analog verfahren.
- y_1, y_2 und y_3 heißen *Konstanten*.

Konstanten und Variablen

- 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 heißt im Kontext imperativer Programmierung *Variable*.
- Im Unterschied zu Konstanten sind Variablen "radierbar".

Konstanten und Variablen

- Die imperative Lösung zur Berechnung von $f(x)$ mit einer Variable y:

  $y = x + 1$
  $y = y + \frac{1}{y}$
  $y = y * y$
- Wenn wir im Folgenden von Konstanten und Variablen sprechen, verwenden wir immer die imperativen Konzepte (außer wir weisen explizit darauf hin), d.h. wir sprechen dann immer von Variablen ("Platzhalter") der Menge V, über die wir Ausdrücke bilden können.
- Variablen sind überschreibbare, Konstanten sind nicht überschreibbare Platzhalter.

Konstanten und Variablen

- Variablen und Konstanten können
  - *deklariert* werden, d.h. ein "leerer Zettel" (Speicherzelle) wird angelegt. Formal bedeutet dies, dass der Bezeichner der Menge der zur Verfügung stehenden Variablen V, die wir in Ausdrücken verwenden dürfen, hinzugefügt wird.
  - *intialisiert* werden, d.h. ein Wert wird auf den "Zettel" (in die Speicherzelle) geschrieben.
    Formal bedeutet dies, dass die Variable mit einem entsprechenden Wert belegt ist.
- Der Wert einer Variablen kann später auch noch durch eine (neue) *Wertzuweisung* verändert werden.
- Die Initialisierung enspricht einer (initialen) Wertzuweisung.
- 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.

Konstanten und Variablen in Java

- 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.
- Das bedeutet: in Java hat jede Variable/Konstante einen Typ.
- Eine Deklaration ist also das Bereitstellen eines "Platzhalters" des entsprechenden Typs.

Konstanten und Variablen 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`**)
- Formal gesehen ist eine Wertzuweisung eine Funktion

$$=: S \to S$$

  mit beliebigem Typ S.
- Damit ist eine Wertzuweisung auch wieder ein Ausdruck.

Abkürzungen für Wertzuweisungen in Java

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`

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

(Auch für weitere Operatoren möglich...)

Konstanten und Variablen in Java

- Konstanten:

```
final <typ> <NAME> = <ausdruck>;
```

```
final double Y_1 = x + 1; //Achtung: was ist x???
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;
```

Zustandsänderungen von Variablen

Beispiel:

Anweisung	x	y	z
`int x;`			
`int y = 5;`		5	
`x = 0;`	0	5	
`final int z = 3;`	0	5	| 3 |
`x += 7 + y;`	12	5	| 3 |
`y = y + z - 2;`	12	6	| 3 |
`x = x * y;`	72	6	| 3 |
`x = x/z;`	24	6	| 3 |

Legende: [ ] = radierbar [\| \|] = nicht radierbar