Kapitel 2

Einfache Rechenstrukturen und Kontrollfluß

Ziele:

- Verstehen der Grunddatentypen von Java
- Verstehen von Typkonversion in Java
- Lernen lokale Variablen und Konstanten zu initialisieren
- Verstehen der Speicherorganisation von lokalen Variablen
- Lernen Fallunterscheidungen zu bilden und iterative Programme zu schreiben
- Lernen Kontrollflußdiagramme als Entwurfsnotation einzusetzen
- Wiederholen der Regeln des Hoare-Kalkül für **`while`**–Programme

Die meisten Programmiersprachen stellen sogenannte Datentypen und deren charakteristische Operationen zur Verfügung. In Java sind dies Datentypen für

- Ganze Zahlen
- Gleitpunktzahlen
- Zeichen
- Boole'sche Werte
- und Felder (später)

Typ	Länge	Wertebereich		
byte	1 Byte	-128	bis	127
short	2 Byte	-32768	bis	32767
int	4 Byte	$-2\,147\,483\,648$	bis	2 147 483 647
long	8 Byte	$-9\,223\,372\,036\,854\,775\,808$	bis	9 223 372 036 854 775 807

Tabelle 2.1: Typen ganzer Zahlen in Java

2.1 Zahlen

Im Rechner werden nur endliche Ausschnitte der Zahlen realisiert.

Ganze Zahlen

Java hat vier Typen ganzer Zahlen, die jeweils 1, 2, 4 und 8 Byte (1 Byte sind 8 Bit) repräsentieren, siehe hierzu Tabelle 2.1. Üblicherweise benützen wir den Typ **int**.

Ganze Zahlen der Typen **byte**, **short**, **int**, **long** (auch **char**) haben eine „2er-Komplement-Darstellung“. Dies bedeutet, daß bei Überschreiten der Obergrenze keine Ausnahme erzeugt wird, sondern im Negativen weitergezählt wird. Für **byte** gilt z.B.

$$\begin{aligned} 127 + 1 &= -128 \\ 127 + 9 &= -120 \\ 127 + 127 &= -2 \end{aligned}$$

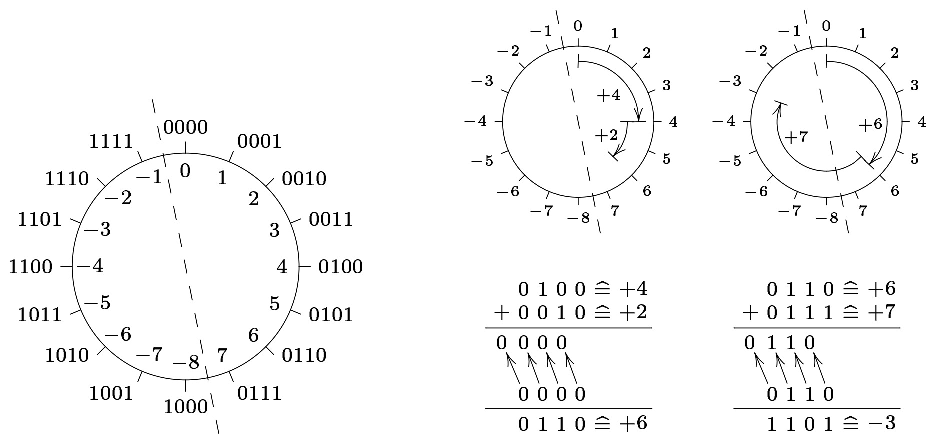
Wir erklären dies im Folgenden am Beispiel von 4-Bit-Zahlen. Einzelheiten lassen sich am Zahlenkreis in Abbildung 2.1 auf der nächsten Seite erkennen. Im Bild 2.1(a) sind die 16 möglichen ganzen Zahlen kreisförmig angeordnet.

Zur Darstellung negativer Zahlen im Zweierkomplement wird der Zahlenkreis entsprechend Abbildung 2.1(a) aufgeteilt. Betragsgleiche Zahlen, z.B. -5 und $+5$, liegen sich waagerecht gegenüber; die Summe ihrer Dualdarstellungen, z.B. also 1011 und 0101, ergibt die Zahlenbasis, hier also: 10000.

In Abbildung 2.1 sind dazu einige Beispiele zusammengestellt. Im Teilbild b sind zwei Beispiele zur Addition zweier positiver Zahlen gezeigt, links ohne, rechts mit Überlauf. Bei einer Addition von Zahlen ungleichen Vorzeichens kann nie ein Überlauf entstehen (Teilbild c). Dagegen kann ein Überlauf bei der Addition zweier negativer Zahlen auftreten (Teilbild d). Ein Überlauf ist also daran feststellbar, daß die Vorzeichen der Summanden untereinander gleich und außerdem ungleich dem Vorzeichen des Ergebnisses sind; die Vorzeichen sind unmittelbar durch das höchstwertige Bit einer Zahl ausgedrückt (siehe z.B. Teilbild a). Alternativ ist ein Überlauf erkennbar an ungleichen Werten der beiden höchsten Stellenüberträge.

Gleitpunktzahlen

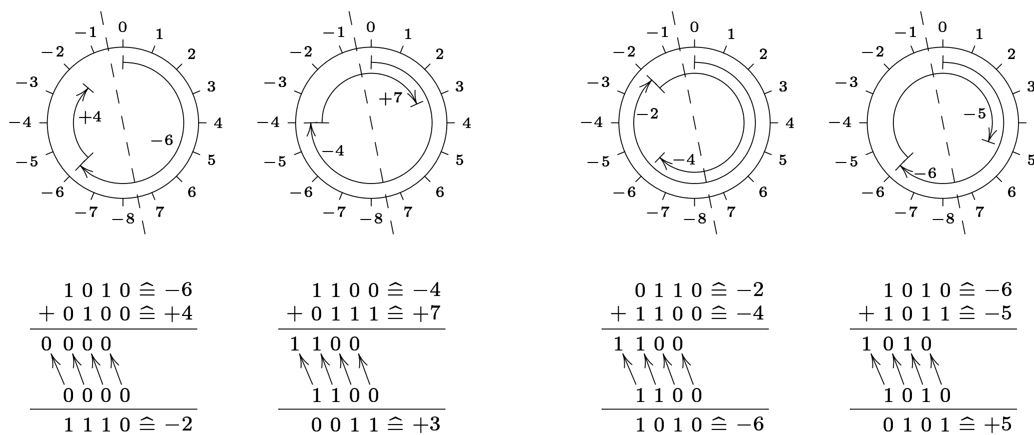
Nach dem IEEE-754–Standard (1985) gibt es zwei Typen von Gleitpunktzahlen, siehe Tabelle 2.2 auf der nächsten Seite. Gleitpunktzahlen des Typs **double** haben die Form



(a) Zahlenkreis mit Zuordnung von Binärworten zu positiven und negativen Dezimalzahlen

Überlauf: NEIN Überlauf: JA

(b) Addition zweier positiver Zahlen



Überlauf: NEIN Überlauf: NEIN Überlauf: NEIN Überlauf: JA

(c) Addition bei Zahlen ungleichen Vorzeichens

(d) Addition zweier negativer Zahlen

Abbildung 2.1: Zahlendarstellung im Zweierkomplement

Typ	Länge	Wertebereich
`float`	4 Byte	bis $\approx 10^{37}$
`double`	8 Byte	bis $\approx 10^{308}$

Tabelle 2.2: Typen von Gleitpunktzahlen in Java

Operator	Bedeutung
*	Multiplikation
/	Division
%	Modulo (Rest)
+	Addition
-	Subtraktion
>	größer
>=	größer oder gleich
<	kleiner
<=	kleiner oder gleich
==	gleich
!=	nicht gleich
=	Zuweisung

Tabelle 2.3: Wichtige Operatoren in Java

$$s \cdot m \cdot 2^{exp}$$

wobei $s \in \{-, +\}$ das Vorzeichen, m eine positive ganze Zahl $< 2^{53}$ die Mantisse und exp eine ganze Zahl zwischen -1075 und 970 den Exponenten darstellt. Beim Typ **`float`** ist $m < 2^{24}$ und exp zwischen -149 und $+104$.

Bemerkung: Diese Darstellung wurde 1937 von Konrad Zuse unter dem Namen halblogarithmische Darstellung eingeführt.

Gleitpunktzahlen schreibt man in Java in der Form

```
double:  6.22,   622E-2,   62.2e-1
 float:  6.22F,  622E-2f,  62.2e-1F
```

Der Exponent wird durch „`e`" oder „`E`" von der Mantisse getrennt. Eine **`float`**-Zahl wird durch das Postfix „`f`" oder „`F`" identifiziert. Üblicherweise ist die Mantisse normalisiert, d.h. die erste Ziffer vor dem Komma ist 1. Beispiel: $-2.5 \mathrel{\widehat{=}} -1 * 1.01 * 2^1$

Arithmetische Operationen

Die wichtigsten arithmetischen Operationen und Vergleichsoperationen sind in Tabelle 2.3 zu finden. Jeder Kasten enthält Operatoren gleicher Präzedenz. Die Operatoren in den oberen Kästen haben höhere Präzedenz als die in den unteren.

Beispiele für Modulo:

$5\%3 = 2$

$5.2\%3 = 2.2$ bzw. $5.2\%2.6 = 0.0$

während für die Division gilt:

$5/3 = 1$

$5.2/3 = 1.733\ldots$ bzw. $5.2/2.6 = 2.0$

$5.2/3.0 = 1.733\ldots$

Typkonversion

Java bringt die zahlartigen Datentypen in folgende „Kleiner-Beziehungen“:

byte < **short** < **int** < **long** < **float** < **double**

In Ausdrücken werden Elemente kleinerer Datentypen automatisch in den größeren Typ konvertiert, falls dies nötig ist.

Beispiele:

`1 + 1.7`	ist vom Typ **double**
`1.0f + 1`	ist vom Typ **float**
`1.0f + 1.0`	ist vom Typ **double**

Umgekehrt ist es möglich durch Voranstellen von `(type)` einen Typ vom Typ `type` zu erzwingen, sofern „`type`“ semantisch korrekt ist. Man nennt dies explizite Konversion (engl. TypeCasting)

Beispiele:

`(byte) 3`	ist vom Typ **byte**
`(int)(2.0 + 5.0)`	ist vom Typ **int**
`(float) 1.3e-7`	ist vom Typ **float**

Typcasting kann auch mit der automatischen (Aufwärts-)Konversion verbunden sein:

`(int) 2.0 + 5.0` ist vom Typ **double**,

da Typcasting stärker bindet als binäre Operatoren.

Bei der Typkonversion kann Information verloren gehen. So können z.B. nach der automatischen Konversion von **int** nach **double** Rundungsfehler auftreten. Bei der expliziten Konversion von Gleitpunktzahlen in ganze Zahlen gehen die Dezimalstellen verloren:

```
(int) 5.2 == 5
(int) -5.2 == -5
```

2.2 Zeichen

Der Typ **char** (für character) bezeichnet die Menge der Zeichen aus dem Unicode-Zeichensatz (mit 16 Bit, d.h. 0–65535 Zeichen, siehe `http://www.unicode.org`). Dieser umfasst den sogenannten ASCII-Zeichensatz mit kleinen und großen Buchstaben, Zahlen und verschiedenen Sonderzeichen. Zeichen werden zur Darstellung von Apostrophen umrahmt. Zeichen können in Integer-Werte und Integer-Werte in die entsprechenden Zeichen des Unicode konvertiert werden.

Operator	Bedeutung
!	strikte Negation
&	strikte Konjunktion („und", auch bitweise Addition)
^	strikte Disjunktion („entweder – oder")
|	strikte Adjunktion („oder")
&&	sequentielle Konjunktion (≘ **andalso** in SML)
||	sequentielle Adjunktion (≘ **orelse** in SML)

Tabelle 2.4: Boole'sche Operationen in Java

Beispiele:

```
(char) 3 == '♡'        (int) '7' == 55
(char) 48 == '0'       (int) 'd' == 100
(char) 100 == 'd'      (int) '!' == 33
```

Bemerkung 1: Obwohl die Elemente von `char` keine Zahlen sind, können sie automatisch in `int` konvertiert werden. Das ist aber schlechter Programmierstil!

Beispiel 2.1 Schlechter Programmierstil

```
int three = 3;
char one = '1';

char four = (char) (three + one);
```

Die Variable `four` in Beispiel 2.1 hat den Typ `char`, obwohl `three + one` den Typ `int` besitzt, der aber durch explizite Konversion in `char` konvertiert wurde.

Bemerkung 2: Der Typ `String` ist kein primitiver Datentyp, sondern ein Objekt. Zeichenketten vom Typ `String` werden in doppelten Anführungszeichen eingeschlossen. Die Konkatenation auf Strings wird mit + notiert. Fast alles wird nach `String` konvertiert:

```
3 + "17" == "317"
```

Konversionen wie `(int) "7"` sind allerdings nicht möglich!

2.3 Boole'sche Werte

Der Typ `boolean` hat genau zwei Werte, `true` und `false`. Die Boole'schen Operationen mit Präzedenzen in absteigender Reihenfolge sind in Tabelle 2.4 aufgeführt. Beispiel 2.2 und Beispiel 2.3 sollen den Unterschied in der Arbeisweise von sequentiellen und strikten boole'schen Operatoren verdeutlichen.

^	true	false
true	false	true
false	true	false

(a) entweder – oder

|, ||	true	false
true	true	true
false	true	false

(b) oder

Abbildung 2.2: Unterschied „entweder – oder“ und „oder“

Beispiel 2.2 Beispiel für die strikte/sequentielle Konjunktion

```
int teiler = 0;
(teiler != 0) && (100/teiler > 1) == false; // Ok
(teiler != 0) & (100/teiler > 1) == false;  // Laufzeitfehler
```

Bemerkung zur Korrespondenz SML – Java

Die Korrespondenzen zwischen den Basistypen von SML und Java sind in Tabelle 2.5 auf der nächsten Seite zu finden. Weitere Unterschiede zwischen SML und Java sind, daß Java sehr viele automatische Typanpassungen vornimmt, die in SML explizit programmiert werden müssen und daß in SML Typinferenz existiert, die es in Java nicht gibt.

2.4 Deklaration lokaler Variablen und Konstanten

Eine einfache Deklaration lokaler Variablen hat die Form

1. `<Type> <VarName>;` bzw.
2. `<Type> <VarName> = <Expression>;`

Der Ausdruck `<Expression>` gibt den Initialwert der Variable `<VarName>` an. Im Fall 2 wird `<VarName>` der Standardwert des Typs `<Type>` als Initialwert zugewiesen. Initialwerte sind z.B.

int, **byte**, **short**	0
float, **double**	0.0f, 0.0

Allerdings dürfen *nur initialisierte* Variablen in Ausdrücken benutzt werden. Deshalb ist es sinnvoll, lokale Variablen sofort zu initialisieren.

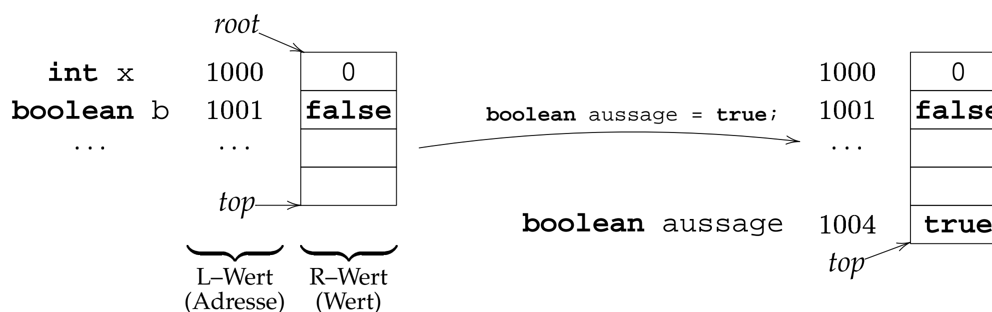
Beispiel 2.3 Beispiel für die strikte/sequentielle Adjunktion

```
true || (1/0 == 1) == true; // Ok
true | (1/0 == 1)           // Laufzeitfehler
```

	Java	SML
Gleitpunktzahlen	**float**, **double**	**real**
unäres Minus	-	~
Division	/	/
Modulo (Rest)	%	nicht vorhanden
Konversion nach ganze Zahl	(**int**)	**truncate**
Ganze Zahlen	**int**	**int**
Ganzzahldiv.	/	**div**
Modulo (Rest)	%	**mod**
Boole'sche Werte	**boolean**	**bool**
strikte Konj.	&	nicht vorhanden
sequ. Konj.	&&	**andalso**
strikte Adj.	|	nicht vorhanden
sequ. Adj.	||	**orelse**
strikte Disj.	^	nicht vorhanden
Worte	String	string
Konkatenation	+	^

Tabelle 2.5: Korrespondenz SML – Java

Lokale Variablen werden im „Keller" (Stack) gespeichert. Durch die Deklaration wird eine neue Speicherzelle für die lokale Variable reserviert und mit ihrem Initialwert belegt. Das folgende Bild zeigt, wie der Speicher bei Deklaration einer lokalen Variable `aussage` um eine Zelle wächst.



Mehrere Deklarationen lokaler Variablen gleichen Typs können zusammengefaßt werden:

```
TypName VarName1 = Ausdruck1, VarName2,
        VarName3 = Ausdruck3;
```

ist eine Abkürzung für

```
TypName VarName1 = Ausdruck1;
TypName VarName2;
TypName VarName3 = Ausdruck3;
```

Beispiel 2.4 Variablendeklaration

```
int total = 17, max = 100, i, j;
```

ist eine Abkürzung für

```
int total = 17;
int max = 100;
int i;
int j;
```

Insgesamt ergibt sich folgende Syntax:

LocalVarDeclaration ::=
Type VarName ["=" Expression] { "," VarName ["=" Expression]} ";"*

In der Java-Spezifikation werden Deklarationen lokaler Variablen folgendermaßen ausgedrückt (vereinfacht):

VariableDeclarator ::=
VarName |
VarName "=" Expression

VariableDeclarators ::=
VariableDeclarator |
VariableDeclarator "," VariableDeclarators

LocalVariableDeclaration ::=
Type VariableDeclarators ";"

Typnamen sind Spezialfälle von „*Type*“. Zum Beispiel ist `String[]` ein *Type*, aber kein Typname.

Eine Konstante wird durch Angabe des „Modifiers“ **`final`** deklariert. Zum Beispiel bezeichnet

```
final int TOTAL = 100;
```

eine Konstante mit Wert 100. Konstanten werden i.a. mit Großbuchstaben geschrieben und sollten (wie auch Variablen) „sprechende“ Namen besitzen. Außerdem sollten in einem Programm Konstanten immer auch als Konstanten deklariert werden und nicht als reine Zahlenwerte gegeben sein. Verwenden Sie *nie* „Magic Numbers“, d.h. Zahlen im Programm, die eine spezielle Bedeutung haben! Beispiele:

- Anstelle von 365 im Programm für „Anzahl der Tage im Jahr“ verwende man besser `final int TAGE_PRO_JAHR = 365;`

- Für die mathematischen Größen π und e verwende man anstelle von 3.14159 und 2.7182 besser `Math.Pi` bzw. `Math.E`

Syntax

ConstantDeclaration ::=
  "`final`" *LocalVariableDeclaration*

Hoare–Regeln

$\{P$ `[exp/x]`$\}$ `type x = exp;` $\{P\}$

Die Hoare–Regel für Konstantendeklarationen und Variablendeklarationen ist die gleiche, wie für Zuweisungen. Außerdem gilt die Abschwächungsregel (für jede Anweisung S)

$$\frac{P \Rightarrow P_1 \quad \{P_1\}\ S\ \{Q_1\} \quad Q_1 \Rightarrow Q}{\{P\}\ S\ \{Q\}} \text{ (Abschwächung)}$$

Beispiele:

1. {`100 == 100`} **`int`** `max = 100;` {`max == 100`}

2.

$$\frac{\begin{array}{c}\texttt{\textbf{true}} \Rightarrow \texttt{100 > 50} \\ \{\texttt{100 > 50}\}\ \texttt{\textbf{int} max = 100;}\ \{\texttt{max > 50}\}\end{array}}{\{\texttt{\textbf{true}}\}\ \texttt{\textbf{int} max = 100;}\ \{\texttt{max > 50}\}}$$

3.

$$\frac{\begin{array}{c}\texttt{max == 40} \Rightarrow \texttt{max-5 = 35} \\ \{\texttt{max-5 == 35}\}\ \texttt{\textbf{final int} C = 5;}\ \{\texttt{max-C ==35}\}\end{array}}{\{\texttt{max == 40}\}\ \texttt{\textbf{final int} C = 5;}\ \{\texttt{max-C == 35}\}}$$

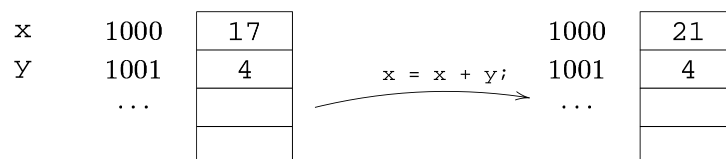
2.5 Zuweisung, sequentielle Komposition und Block

Eine Zuweisung wird in Java mit „=“ geschrieben (im Gegensatz zu Pascal, Modula etc., wo „`:=`“ verwendet wird).

```
<VarName> = <Ausdruck>;
```

bedeutet, daß der Variablen mit Name `<VarName>` der Wert von `<Ausdruck>` als R–Wert zugewiesen wird.

Beispiel:



Bemerkung: Später werden wir sehen, daß auf der rechten Seite der Zuweisung allgemeinere Ausdrücke stehen können.

Für manche spezielle Zuweisungen gibt es Abkürzungen, die von der Programmiersprache C übernommen wurden. Sei `x` eine Variable von numerischem Typ und `op` eine binäre Operation. Dann gibt es folgende Abkürzungen:

x++;	steht für	x = x + 1;
x--;	steht für	x = x - 1;
x op= <Ausdruck>;	steht für	x = x op <Ausdruck>;

Beispiele:			
	x += y;	steht für	x = x + y;
	x &&= c;	steht für	b = b && c;
	x += 3*y;	steht für	x = x + 3*y;

Syntax

Assignment ::=
Name "=" Expression ";"

Hoare–Regel (Wiederholung)

$$\{P[\texttt{exp/x}]\}\ \texttt{x = exp;}\ \{P\}$$

Beispiel:

1. $\{\texttt{max - 10 > 50}\}$ `max = max - 10;` $\{\texttt{max > 50}\}$
2. Aus `max = 100` folgt wegen `max = 100` $\Rightarrow$ `max - 10 > 50` mit der Abschwächungsregel

$$\{\texttt{max = 100}\}\ \texttt{max = max - 10;}\ \{\texttt{max > 50}\}$$

3. $\{\texttt{max - C = 35}\}$ `max = max - C;` $\{\texttt{max = 35}\}$

Sequentielle Komposition in Java wird durch Hintereinanderschreiben ausgedrückt:

Statements ::=
Statement | Statement Statements

wobei bisher

Statement ::=
LocalVariableDeclaration | ConstantDeclaration | Assignment

Doppeldeklarationen von Variablen sind *nicht* erlaubt!

Hoare-Regel

$$\frac{\{P\}\ S_1\ \{R\} \qquad \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1\, S_2\ \{Q\}}$$

Beispiel 2.5 Sequentielle Komposition in Java

```java
int total = 100;
total = total + 100;
```

Beispiel:

$$\frac{\{\texttt{max == 40}\}\ \textbf{final int}\ \texttt{C = 5;}\ \{\texttt{max-C == 35}\} \quad \{\texttt{max-C == 35}\}\ \texttt{max = max - C;}\ \{\texttt{max == 35}\}}{\{\texttt{max == 40}\}\ \textbf{final int}\ \texttt{C = 5; max = max - C;}\ \{\texttt{max == 35}\}}$$

Mehrere Anweisungen können durch geschweifte Klammern zu einem *Block* zusammengefügt werden.

BlockStatement ::=
"{"Statements "}"

wobei ein Block wieder selbst als Anweisung betrachtet wird:

Statement ::= BlockStatement

Der Gültigkeitsbereich einer lokalen Variablen oder Konstante ist der die Deklaration umfaßende Block. Außerhalb dieses Blocks existiert die Variable *nicht*! Geschachtelte Doppeldeklarationen sind (im Gegensatz zu Modulen) verboten.

Beispiel 2.6 Gültigkeitsbereiche

```java
{
    int wert = 0;
    wert = wert + 17;
    {
        int total = 100;
        wert = wert - total;
    }
    wert = 2 * wert;
}
```

Der Gültigkeitsbereich von `wert` in Beispiel 2.6 erstreckt sich von Zeile 2 bis Zeile 9, derjenige von `total` von Zeile 5 bis 7.

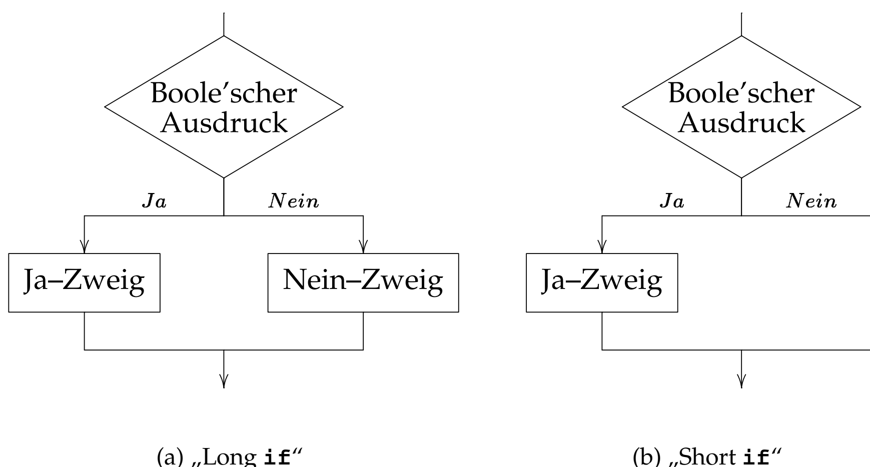
Hoare-Regel

$$\frac{\{P\}\ S\ \{Q\}}{\{P\}\ \{S\}\ \{Q\}}$$

falls P, Q keine lokalen Variablen von S enthalten.

Beispiel:

$$\frac{\{\texttt{max == 40}\}\ \textbf{final int}\ \texttt{C = 5; max = max - C;}\ \{\texttt{max == 35}\}}{\{\texttt{max == 40}\}\ \{\textbf{final int}\ \texttt{C = 5; max = max - C;}\}\ \{\texttt{max == 35}\}}$$



(a) „Long **if**“ (b) „Short **if**“

Abbildung 2.3: Kontrollflußdiagramm für **if**

2.6 Fallunterscheidung und Kontrollflußdiagramme

Die Fallunterscheidung in Java hat die Form

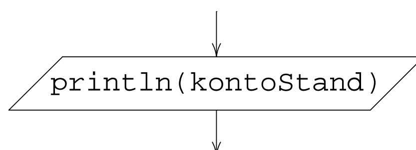
if (Boolescher Ausdruck) *Statement*

bzw.

if (Boolescher Ausdruck) *Statement* **else** *Statement*

In graphischer Notation schreibt man den Kontrollfluß einer Fallunterscheidung wie in Abbildung 2.3 zu sehen ist.

Ein Kontrollflußdiagramm dient zur Beschreibung des Ablaufs eines Programms. Eine Fallunterscheidung wird durch eine Raute repräsentiert, die die Bedingung enthält. Jede Anweisung wird in einen rechteckigen Kasten geschrieben. Ein- und Ausgaben werden in Parallelogramme geschrieben. Beispiel:



Jeder Pfeil repräsentiert die sequentielle Abfolge eines Ablaufs. Pfeile können mit „Ja“ oder „Nein“ annotiert sein. Siehe hierzu auch die Beispiele 2.7 auf der nächsten Seite und 2.8 auf der nächsten Seite (aus dem Bankbereich).

Falls mehrere Anweisungen in der Fallunterscheidung vorkommen sollten, faßt man diese in einem Block zusammen. Die Blockklammern in Beispiel 2.9 sind sehr wichtig! Will man z.B. auch im **else**–Zweig mehrere Anweisungen verwenden und

Beispiel 2.7 Kontrollfluß 1

```
if (kontoStand >= betrag)
    kontoStand = kontoStand - betrag;
```

Beispiel 2.8 Kontrollfluß 2

```
if (kontoStand >= betrag)
    kontoStand = kontoStand - betrag;
else
    kontoStand = kontoStand - betrag - UEBERZIEH_GEBUEHR;
```

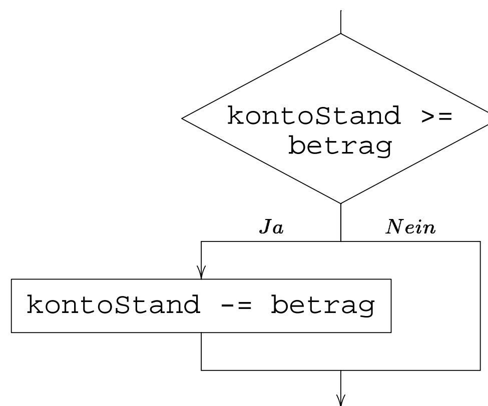


Abbildung 2.4: Kontrollfluß für Beispiel 2.7

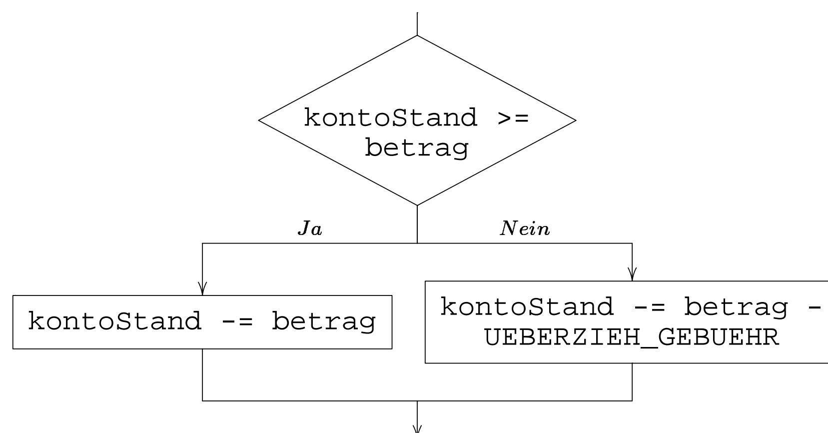


Abbildung 2.5: Kontrollfluß für Beispiel 2.8

Beispiel 2.9 mehrere Anweisungen in einer Fallunterscheidung

```
if (kontoStand >= betrag)
{
    double neuerStand = kontoStand - betrag;
    kontoStand = neuerStand;
}
else
{
    kontoStand = kontoStand - betrag - UEBERZIEH_GEBUEHR;
}
```

Beispiel 2.10 mehrere Anweisungen II

```
if (kontoStand >= betrag)
{
    double neuerStand = kontoStand - betrag;
    kontoStand = neuerStand;
}
else
    kontoStand = kontoStand - betrag - UEBERZIEH_GEBUEHR;
    gebuehren += UEBERZIEH_GEBUEHR;
```

vergißt die Blockklammer, so erhält man meist falsche Ergebnisse, wie dies in Beispiel 2.10 zu sehen ist. Hier wird die letzte Zeile ausgeführt, auch wenn der Kontostand den abgehobenen Betrag überschreitet! Deshalb muß geklammert werden, die nötige Korrektur ist in Beispiel 2.11 auf der nächsten Seite zu finden.

Bemerkung: Falls das Programm später geändert wird, ist empfohlen, die Blockklammern immer zu verwenden.

Mehrere Fallunterscheidungen hat Beispiel 2.12 auf der nächsten Seite. Hier ist es wichtig, „**`else`**“ zu verwenden, da z.B. für `leistung == 400` (MHz) alle drei ersten Alternativen zutreffen!

Das Problem des „Dangling `else`“

In Java wird eine Fallunterscheidung nicht durch explizite Begrenzungssymbole abgeschlossen. Dadurch können sich Doppeldeutigkeiten ergeben, wie das folgende Beispiel zeigt.

Sei der Kontrollfluß zur Berechnung einer Seminargebühr aus Abbildung 2.6 auf Seite 29 gegeben.

Bemerkung: Dies ist kein guter Entwurf. Besser wäre eine direkte Fallunterscheidung nach „Student?“ wie auf Abbildung 2.7 auf Seite 30.

Um die korrekte Fallunterscheidung zu erzielen, muß man klammern. Bei Java wird immer das **`else`** zum nächsten **`if`** gebunden, d.h.

Beispiel 2.11 mehrere Anweisungen III

```
if (kontoStand >= betrag)
{
    double neuerStand = kontoStand - betrag;
    kontoStand = neuerStand;
}
else
{
    kontoStand = kontoStand - betrag - UEBERZIEH_GEBUEHR;
    gebuehren += UEBERZIEH_GEBUEHR;
}
```

Beispiel 2.12 Prozessorleistung

```
if (leistung < 200)
    System.out.println(leistung + " MHz ist zu langsam");
else if (leistung < 350)
    System.out.println(leistung + " MHz schnell genug");
else if (leistung <= 550)
    System.out.println(leistung + " MHz ist sehr schnell");
else
    System.out.println("Hat Ihr System wirklich die Leistung " +
        leistung + " MHz?");
```

Beispiel 2.13 Seminargebühr – fehlerhaft

```
double gebuehr = 10.00; // 10 DM fuer Studenten mit Nebenfach Informatik
if (student)
    if (hauptfach = INFORMATIK)
        gebuehr = 5.00; // Informatikstudenten haben Rabatt
else                    // Problem: Nein-Zweig bezieht sich auf inneres if!
    gebuehr = 15.00;
```

Beispiel 2.14 Seminargebühr – korrigiert

```
double gebuehr = 10.00;
if (student)
{
    if (hauptfach = INFORMATIK)
        gebuehr = 5.00;
}
else
    gebuehr = 15.00;
```

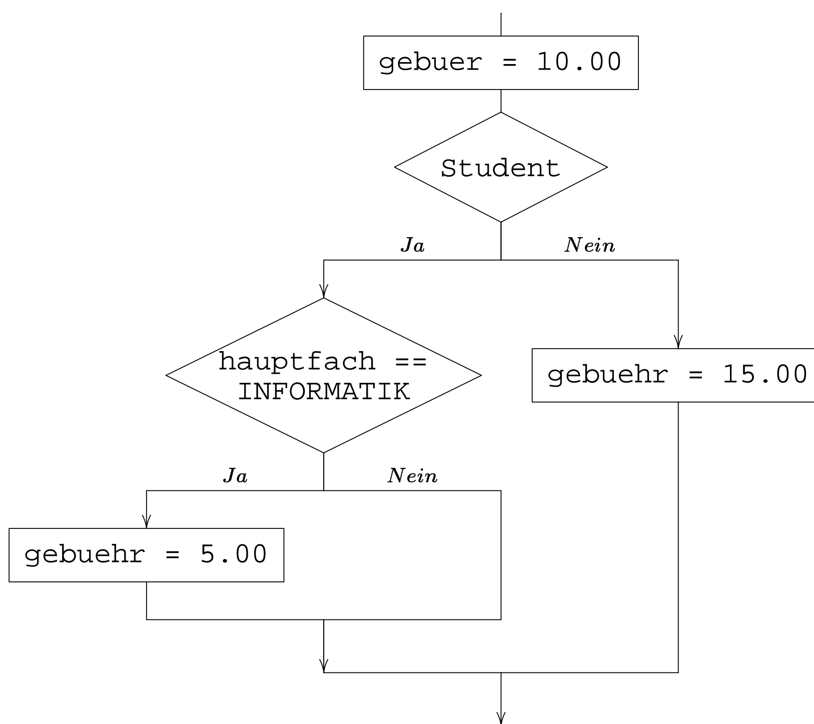


Abbildung 2.6: Kontrollfluß zu einer Seminargebühr

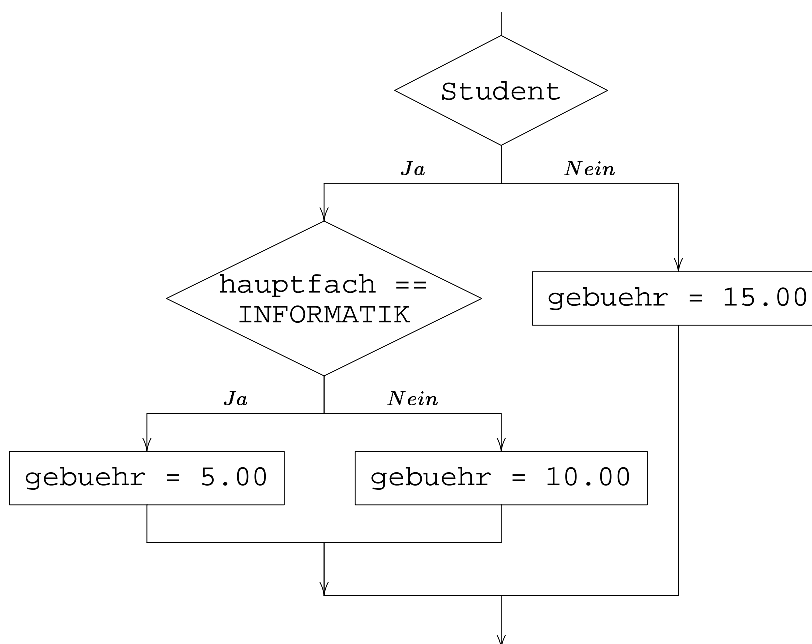


Abbildung 2.7: Kontrollfluß zu einer Seminargebühr – verbessert

```
if (c1) if (c2) S1 else S2
```

und

```
if (c1) { if (c2) S1 else S2 }
```

sind äquivalent. Innerhalb einer Fallunterscheidung ist kein „kurzes **`if`**“ (**`if`** ohne **`else`**–Zweig) als Anweisung erlaubt!

Syntax

ShortIfStatement ::=
    "`if`" "`(`" *Expression* "`)`" *Statement*[1]
IfStatement ::=
    ShortIfStatement "`else`" *Statement*[1]

wobei Expression vom Typ **`boolean`** ist. Dadurch wird *Statement* erweitert durch

Statement ::=
    ShortIfStatement | *IfStatement*

Hoare-Regeln

$$\frac{\{\texttt{b \& } P\}\ S_1\ \{Q\} \qquad \{\texttt{(!b) \& } P\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ \texttt{(b)}\ S_1\ \textbf{else}\ S_2\ \{Q\}} \quad \text{(if)}$$

[1]hier darf *Statement kein ShortIfStatement* sein!

Zur übersichtlichen Darstellung von Beweisen verwenden wir sogenannte *Beweisskizzen*. In einem Programm wird jede Anweisung mit einer Vor- und einer Nachbedingung annotiert. Solche Zusicherungen können mit „$\Downarrow$" abgeschwächt werden.

Beispiele:

1. Zuweisung

```
{true}
   ⇓
{x * x >= 0}
 x = x * x;
  {x >= 0}
```

2. if-then-else-Regel

```
{x == A}
if (x >= 0)
    {x >= 0 & x == A}
            ⇓
    {x == A & x == |A|}
    y = x;
    {x == A & y == |A|}
else
    {x < 0 & x == A}
            ⇓
    {x == A & -x == |A|}
    y = -x;
    {x == A & y == |A|}
{x == A & y == |A|}
```

Hier ist A eine „logische Variable", die nicht im Programm vorkommt.

2.7 Iteration

In Java (wie in den meisten imperativen Systemen) gibt es drei Konstrukte zur Iteration: **while**-, **for**- und **do**-Schleifen.

2.7.1 while-Schleifen

Die **while**-Schleife hat die Form

```
while ( Boolescher Ausdruck )
    Statement
```

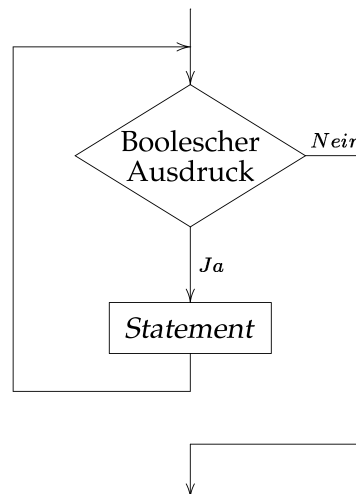


Abbildung 2.8: Kontrollflußdiagramm für **while**

wobei *Statement* meist ein Block ist. Solange der Boolesche Ausdruck den Wert **true** hat, wird das Statement ausgeführt. Das Kontrollflußdiagramm ist in Abbildung 2.8 zu finden, Beispiele für **while** sind Beispiel 2.15 und Beispiel 2.16 auf der nächsten Seite.

Beispiel 2.15 10mal „tick“ drucken

```
int n = 1, end = 10;
while (n <= end)
{
    System.out.println("tick" + n);
    n++;
}
```

Syntax

WhileStatement ::=
"while" "(" *Expression* ")" *Statement*

wobei *Expression* vom Typ **boolean** ist. Innerhalb eines **while** darf Statement kein „kurzes **if**“ sein (sonst besteht die Gefahr eines Dangling **else**).

Hoare-Regeln

partielle Korrektheit:

$$\frac{\{\texttt{b \& } I\}\ S\ \{I\}}{\{I\}\ \textbf{while}\ \texttt{(b)}\ S\ \{\texttt{(!b) \& } I\}}\ \text{(Iteration)}$$

Hier ist I eine sog. *Invariante*.

Beispiel 2.16 Quersumme von x

```
int qs = 0, x = 352;
while (x > 0)
{
    qs = qs + x % 10;
    x = x / 10;
}
```

totale Korrektheit:

$$\frac{\begin{array}{c}\{\texttt{b \& } I\}\ S\ \{I\} \\ \{\texttt{b \& } I \texttt{ \& t == z}\}\ S\ \{\texttt{t < z}\} \\ I \Rightarrow \texttt{t >= 0}\end{array}}{\{I\}\ \textbf{while}\ \texttt{(b)}\ S\ \{\texttt{(!b) \& } I\}}\ (\text{Iteration}_{tot})$$

wobei `t` ein Integer-Ausdruck ist und `z` eine logische Variable, die nicht in I, `b`, `s` oder `t` vorkommt.

Beispiel
Für das `while`-Programm

$$P := \textbf{while}\ \texttt{(x > 0) \{ y = y + 1; x = x - 1; \}}$$

zeigen wir die totale Korrektheit bezüglich der Vorbedingung

`x >= 0 & x == a & y == b`

und der Nachbedingung

`y == a + b & x == 0.`

Beweis. Wir wählen `x + y == a + b & x >= 0` als Invariante I.

1. Beweis der partiellen Korrektheit bzgl. der Invariante I

   a) Es gilt für den Schleifenrumpf

```
{x + y == a + b & x >= 0 & x > 0}
              ⇓
{x -1 + y + 1 == a + b & x - 1 >= 0}
    y = y + 1;
{x - 1 + y == a + b & x - 1 >= 0}
    x = x - 1;
{x + y == a + b & x >= 0}
```

b) Mit der **`while`**-Regel für partielle Korrektheit folgt daraus

```
{x + y == a + b & x >= 0}
            P
{x + y == a + b & x >= 0 & !(x > 0)}
```

2. Durch Abschwächung erhalten wir aus 1b:

   ```
   {x == a & y == b & x >= 0} P {x == a + b & x == 0}
   ```

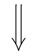
   Also gelten die gewünschten Vor- und Nachbedingungen.

3. Zum Beweis der totalen Korrektheit muß noch die Terminierung gezeigt werden. `x` ist die einzige lokale Variable im Programm; wir definieren `t` := `x`. Dann gilt offensichtlich:

   a) `x + y == a + b & x >= 0` $\Longrightarrow$ `x >= 0`, d.h. $I \Rightarrow$ `t >= 0`

   b)
   ```
   {x >= 0 & x + y == a + b & x >= 0 & x == z}
       y = y + 1;
       x = x - 1;
   {x < z}
   ```

   Beweis von 3b:

   ```
   {...& x == z}
   ```
   
   ```
   {x - 1 < z}
       y = y + 1;
   {x - 1 < z}
       x = x - 1;
   {x < z}
   ```

□

2.7.2 `for`-Schleifen

Die häufigste Form der **`while`**-Schleife ist

```
i = start;
while (i < end)
{
    ...
    i++
}
```

Dies kann durch eine **for**-Schleife abgekürzt werden:

```
for (i = start; i <= end; i++)
{
    ...
}
```

Beispiel 2.17 „tick“ mit einer **for**-Schleife

```
int end = 10;
for (int n = 1; n <= end; n++)
{
    System.out.println("tick" + n);
}
```

Allgemein hat eine **for**-Schleife die Gestalt

for (Initialisierung; Bedingung; Zählerkorrektur;)
Statement

Dabei wird zunächst die Initialisierung ausgeführt. Dann wird *Statement* ausgeführt und der Zähler geändert, solange die Bedingung wahr ist. Guter Stil ist es, **for**-Schleifen nur folgendermaßen zu schreiben:

```
for (setze counter auf start;
     Test, ob counter bei end;
     aendere counter)
{
    ... // counter, start, end und increment werden
        // hier nicht geaendert!
}
```

Außerdem ist es sinnvoll, den Zähler `counter` in der Initialisierung zu deklarieren. Dann ist `counter` lokal für die **for**-Anweisung und außerhalb nicht definiert.

2.7.3 do-Schleifen

Die **do**-Schleife ist eine **while**-Schleife, bei der die Anweisung mindestens einmal aufgeführt wird. Die Bedingung wird erst nach Ausführung der Anweisung überprüft. Sie hat die Form

do
Statement
while (Boolescher Ausdruck)

Das Kontrollflußdiagramm ist in Abbildung 2.9 auf der nächsten Seite zu finden.

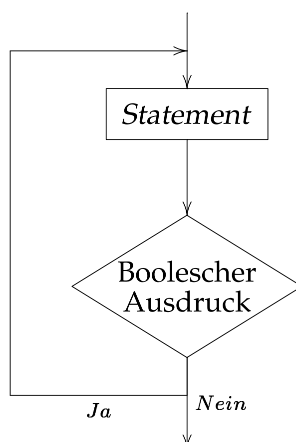


Abbildung 2.9: Kontrollflußdiagramm für **`do`**

2.8 Zusammenfassung

1. Java besitzt 4 Grunddatentypen für ganze Zahlen (**`byte`**, **`short`**, **`int`**, **`long`**) und 2 Grunddatentypen für Gleitpunktzahlen (**`float`**, **`double`**). Dazu kommen noch **`boolean`** und **`char`**. `String` ist *kein* Grunddatentyp.

2. Java hat eine automatische Konversion in den „größeren“ Grunddatentyp. Konversion in einen „kleineren“ Datentyp geschieht explizit durch Typcasting.

3. Eine Fallunterscheidung erlaubt es, abhängig von einer Bedingung, verschiedene Anweisungen auszuführen.

4. Zur Vermeidung des Dangling-**`else`**-Problems dürfen „Ja“- und „Nein“-Zweig von **`if`** sowie die Anweisung von **`while`** kein „Short-**`if`**“ sein.

5. Eine Iteration (Schleife) dient zur mehrfachen Ausführung eines Blocks von Anweisungen. Die Terminierungsbedinung kontrolliert, wie häufig der Block ausgeführt wird.

6. Es gibt 3 Arten von Iterationen: **`while`**-, **`for`**- und **`do`**-Schleifen. **`for`**-Schleifen sollten verwendet werden, wenn die Schleifenvariable von einem Anfangswert bis zu einem Endwert mit einem *konstanten* Inkrement oder Dekrement läuft; **`do`**-Schleifen sind passend, wenn der Schleifenrumpf mindestens einmal ausgeführt werden muß.

7. Kontrollflußdiagramme dienen zur graphischen Darstellung von Programmabläufen. Sie können sehr gut zum Entwurf und zur Veranschaulichung iterativer Programme eingesetzt werden.

8. Der Hoare-Kalkül erlaubt den Beweis der partiellen und totalen Korrektheit (kleiner) Programme.