

Iteration

- Die `while` Schleife
- Die `for` Schleife
- Schleifeninvarianten und Hoare Logik
- Dateiein- und ausgabe
- Simulationen

Die While-Schleife

$$\begin{aligned} \langle \textit{statement} \rangle & ::= \dots \\ & | \text{ while } (\langle \textit{expression} \rangle) \langle \textit{statement} \rangle \\ & \dots \end{aligned}$$

Ausführung von `while (e) c`:

Das Statement `c` wird solange immer wieder ausgeführt, bis der Ausdruck `e` den Wert `true` hat.

Der Ausdruck `e` muss vom Typ `Boolean` sein.

Der Ausdruck `e` wird vor jeder Ausführung von `c` ausgewertet. Ist er schon zu Beginn `true`, so wird `c` überhaupt nicht ausgeführt.

Wird er niemals `true`, so wird `c` immer wieder ausgeführt (Endlosschleife).

Beispiel: Zins und Zinseszins

Ein Cent werde zu 3% p.a. angelegt. Nach wieviel Jahren ist das Kapital auf 100€ angewachsen?

Wir müssen solange 3% dazuzählen, bis 100€ erreicht sind und gleichzeitig die Zahl der Durchläufe in einer lokalen Variablen speichern.

Lösung

```
public class Investment {
    public static void main(String[] args) {
        double kapital = 0.01;
        int years = 0;
        while (kapital < 100.) {
            years = years + 1;
            kapital = kapital * 1.03;
        }
        System.out.println("Es dauert " + years + " Jahre");
    }
}
```

Dasselbe in OCAML (ohne Ausgabe):

```
let rec dauer jahre kapital =
  if kapital >= 100. then jahre
  else dauer (jahre+1) (kapital *. 1.03) in
dauer 0 0.01
```

While durch Rekursion

Betrachte den Spezialfall einer While-Schleife:

$$\text{typ}_1 x_1 = e_1 i$$

...

$$\text{typ}_n x_n = e_n i$$
$$\text{while}(b) \{$$
$$x_1 = e'_1 i$$

...

$$x_n = e'_n i$$
$$\}$$

Der Wert eines Ausdrucks r *nach* Ausführung der Schleife kann wie folgt rekursiv berechnet werden:

While durch Rekursion

```
let rec f  $x_1 \cdots x_n$  =  
if not  $c$  then  $r$  else  
  let  $x_1 = e'_1$  in  
  ...  
  let  $x_n = e'_n$  in  
  f  $x_1 \cdots x_n$ 
```

Schnelles Potenzieren

Folgendes Programmstück berechnet die Potenz a^n (Ergebnis ist in r).

```
double r = 1;
double b = a;
int i = n;

while (i > 0) {
    if (i % 2 == 1) {
        r = r * b;
    }
    b = b * b;
    i = i / 2;
}
```

Beispiel 3⁵

```
double r = 1; double b = a; int i = n;
while (i > 0) {
    if (i % 2 == 1)
        r = r * b;
    b = b * b; i = i / 2; }
```

b	r	i
3	1	5
9	3	2
81	3	1
81 ²	243	0

Ordentliche Begründung

Wie funktioniert das ?

Es basiert auf den Gleichungen $x^{2i+1} = x^{2i}x$ und $x^{2i} = (x^i)^2$.

Wie formalisiert man diese Idee ?

Invariante

Wir bezeichnen mit I die Eigenschaft, dass $r \cdot b^i = a^n$.

Die Eigenschaft I gilt zu Beginn der Schleife.

Gilt sie vor Ausführung des Schleifenrumpfes, so gilt sie auch danach.
(Begründung folgt.)

Somit gilt sie bei Verlassen der Schleife.

Da dann $i=0$ ist, folgt $r = a^n$.

Begründung

Es bezeichne r_{alt} , r_{neu} den Wert von r vor und nach dem Schleifenrumpf.
Ebenso für b , i .

Falls i_{alt} gerade ist, so gilt:

$$b_{\text{neu}} = b_{\text{alt}}^2$$

$$i_{\text{neu}} = i_{\text{alt}}/2$$

$$r_{\text{neu}} = r_{\text{alt}}$$

Somit $r_{\text{neu}} b_{\text{neu}}^{i_{\text{neu}}} = r_{\text{alt}} b_{\text{alt}}^{i_{\text{alt}}}$.

Begründung

Falls i_{alt} ungerade ist, so gilt:

$$b_{\text{neu}} = b_{\text{alt}}^2$$

$$i_{\text{neu}} = (i_{\text{alt}} - 1)/2$$

$$r_{\text{neu}} = r_{\text{alt}} \cdot b_{\text{alt}}$$

Wir rechnen: $r_{\text{neu}} b_{\text{neu}}^{i_{\text{neu}}} = r_{\text{alt}} \cdot b_{\text{alt}} \cdot b_{\text{alt}}^{2i_{\text{neu}}} = r_{\text{alt}} \cdot b_{\text{alt}} \cdot b_{\text{alt}}^{i_{\text{alt}}-1} = r_{\text{alt}} b_{\text{alt}}^{i_{\text{alt}}}$.

Die Eigenschaft I heißt **Invariante**.

Es bietet sich oft an, bei Schleifen nach geeigneten Invarianten zu suchen.

Hoare Logik

Ein Hoare-Tripel ist ein formaler Ausdruck der Form

$$\{P\}c\{Q\}$$

wobei P, Q Aussagen über den Programmzustand sind (Aussehen des Heaps, Werte von Variablen) und c ein Statement ist.

Die P, Q werden als *Zusicherungen* bezeichnet.

Ein solches Hoare Tripel ist *gültig*, wenn für jeden Programmzustand q , der P (die *Vorbedingung*) erfüllt, folgendes gilt: falls die Abarbeitung von c ausgehend von diesem Zustand q terminiert in einem Folgezustand q' , so muss dieser Folgezustand q' die Zusicherung Q (*Nachbedingung*) erfüllen.

Wichtig: Für unsere Zwecke sind “Aussagen” deutsche oder englische Sätze, die informelle Mathematik- und Logiknotation verwenden dürfen.

Will man formale Programmverifikation betreiben, so muss man sich auf eine formalisierte *Zusicherungssprache* verständigen.

Die Hoare'schen Regeln

Es gibt nun einen Satz von (syntaxgerichteten) Regeln, die es gestatten, die gültigen Hoare Tripel formal herzuleiten.

Hier ist die Hoare-Regel für die While Schleife.

$$\frac{\{I \wedge b\}c\{I\} \quad P \rightarrow I \quad I \wedge \neg b \rightarrow Q}{\{P\}\text{while}(b)c\{Q\}} \quad (\text{H-WHILE})$$

Hierbei setzen wir vereinfachend voraus, dass die Auswertung der Bedingung b keine Seiteneffekte hat. Kommt die Bedingung in einer Zusicherung vor, so ist der Wahrheitswert der Bedingung in dem Zustand, auf den sich die Zusicherung bezieht, gemeint.

Erklärung der Regel

Um zu zeigen, dass $\{P\}\text{while}(b)c\{Q\}$ gültig ist, muss eine geeignete Zusicherung I , die Invariante, gefunden werden, derart dass,

1. P impliziert I (für beliebigen Zustand)
2. $\{I \wedge b\}c\{I\}$ ist gültiges Hoare Tripel (ggf. vermöge anderer Hoare'scher Regeln)
3. $I \wedge \neg b$ impliziert Q . D.h. jeder Zustand q , der I erfüllt und in dem b den Wert `false` hat, erfüllt die Zusicherung Q .

Regel für die Zuweisung

Sei e ein seiteneffektfreier Ausdruck, x eine Variable.

Es gilt die folgende Hoare Regel für das Zuweisungsstatement $x=e$:

$$\frac{P \rightarrow Q[x := e]}{\{P\}x = e\{Q\}} \quad (\text{H-Ass})$$

Hier bedeutet $Q[x := e]$ die Zusicherung, welche in einem Zustand q erfüllt ist, genau dann, wenn Q in dem Zustand q' erfüllt ist, welcher sich von q nur dadurch unterscheidet, dass x den Wert von e in q hat.

Ist z.B.: $Q \equiv "x = 19"$, also in all den Zuständen erfüllt, in denen x den Wert 19 hat, so ist $Q[x := (x+1)] \equiv "x + 1 = 19"$, also " $x = 18$ ".

In der Tat gilt z.B.: $\{x = 18\}x=x+1\{x = 19\}$.

Regel für die Hintereinanderausführung

$$\frac{\{P\}c_1\{R\} \quad \{R\}c_2\{Q\}}{\{P\}c_1 ; c_2\{Q\}} \quad (\text{H-SEQ})$$

Hier bezeichnet $c_1 ; c_2$ die Hintereinanderausführung von c_1, c_2 . In Java müsste man das strenggenommen als $\{c_1 c_2\}$ schreiben.

Es hat den Anschein, als müsste man auch hier bei der Rückwärtsanwendung die Zusicherung R geschickt “raten”.

Das ist nicht der Fall: Man wählt vielmehr R als die schwächste Bedingung, sodass $\{R\}c_2\{Q\}$ noch gilt, z.B.: $R \equiv Q[x := e]$ im Falle $c_2 \equiv x=e$.

Man kann diese Idee so systematisieren, dass eine Herleitung in der Hoare-Logik aus geeigneten Invarianten für die Schleifen automatisch erzeugt werden kann. Die Details werden hier nicht behandelt.

Regel für die Fallunterscheidung

Sei b ein seiteneffektfreier Ausdruck vom Typ Boolean. Es gilt die folgende Hoare Regel:

$$\frac{\{P \wedge b\}c_1\{Q\} \quad \{P \wedge \neg b\}c_2\{Q\}}{\{P\}\text{if}(b) c_1 \text{ else } c_2\{Q\}} \quad (\text{H-IF})$$

Weiterführendes

- Man kann zeigen, dass alle gültigen Hoare-Tripel, die die behandelten Konstrukte (while, if, Zuweisung) enthalten, auch mit den Hoare-Regeln hergeleitet werden können.
- Es gibt in der Literatur andere äquivalente Formulierungen der Hoare-Regeln.
- Es gibt Hoare-Regeln für alle anderen Java Konstrukte, insbesondere Methodenaufrufe, auch rekursiv, sowie für Ausdrücke mit Seiteneffekten.
- Es gibt Versionen der Hoare Logik, bei denen in der Nachbedingung ein Zugriff auf die Werte der Variablen vor Ausführung des Statements möglich ist. Dieser Effekt kann in unserer Version nur über logische Variablen erreicht werden: $\{x = A\}c\{x = A\}$ drückt aus, dass c den Wert von x nicht verändert.

Erfinder der Hoare Logik



C.A.R. “Tony” Hoare (1934–)

$$\frac{\{I \wedge b\} c \{I\} \quad P \rightarrow I \quad I \wedge \neg b \rightarrow Q}{\{P\} \text{while}(b) c \{Q\}}$$

$$\frac{P \rightarrow Q[x:=e]}{\{P\} x = e \{Q\}}$$

$$\frac{\{P\} c_1 \{R\} \quad \{R\} c_2 \{Q\}}{\{P\} c_1 ; c_2 \{Q\}}$$

$$\frac{\{P \wedge b\} c_1 \{Q\} \quad \{P \wedge \neg b\} c_2 \{Q\}}{\{P\} \text{if}(b) c_1 \text{ else } c_2 \{Q\}}$$

For-Schleife

Oft muss ein Statement eine bestimmte, feste Zahl von Malen durchlaufen werden.

```
int summe = 0;
for(int i = 0; i <= 100; i = i + 1) {
    summe = summe + i;
}
```

Jetzt ist summe gleich 5050.

Formal

$\langle \textit{statement} \rangle ::= \dots \mid \text{for} (\langle \textit{expression} \rangle ; \langle \textit{expression} \rangle ; \langle \textit{expression} \rangle) \langle \textit{statement} \rangle$

Ausführung von $\text{for} (\textit{init} ; \textit{cond} ; \textit{step}) \textit{body}$:

init und *step* sind *Ausdrücke* mit Seiteneffekten (Zuweisungen oder Methodenaufrufe).

cond ist ein Ausdruck vom Typ Boolean. *body* ist ein beliebiges Statement.

Ausgeführt wird:

```
init ;  
while ( cond ) {  
    rumpf  
    step ;  
}
```

Das Ausdrucks-Statement

Ist e ein beliebiger Ausdruck, so ist

$$e ;$$

ein Statement.

Der Ausdruck e wird ausgewertet und sein Ergebnis verworfen. Das macht natürlich nur Sinn, wenn e Seiteneffekte hat.

Das Zuweisungsstatement

$$x = e ;$$

ist formal ein Spezialfall des Ausdrucksstatements, da $x = e$ ein Ausdruck ist: Seine Auswertung weist als Seiteneffekt den Wert von e der Variablen x zu. Wert dieses Ausdrucks ist der Wert von e .

Eine Zeichenkette umdrehen

In der Klasse `String` gibt es die Methode `length` und `charAt`.

Man verwendet sie so:

```
"Matthias".length() ist 8
```

```
"Matthias".charAt(2) ist 't'
```

Die Methode `charAt` liefert ein Ergebnis vom Typ `char`. Man kann `chars` mit `+` an strings anhängen.

Wir wollen jetzt einen beliebigen String `s` umdrehen, also aus `Matthias` soll `saihttaM` werden.

Dazu müssen wir `s` der Reihe nach durchgehen und die einzelnen Zeichen in umgekehrter Reihenfolge aneinanderhängen.

Lösung

```
String t = "";  
for (int i = 0 ; i < s.length() ; i++) {  
    t = s.charAt(i) + t;  
}
```

Wir können den String auch vom Ende her durchgehen:

```
String t = "";  
for (int i = s.length()-1 ; i >=0 ; i--) {  
    t = t + s.charAt(i) ;  
}
```

Merke: Im Rumpf einer While oder For-Schleife ist die Bedingung **immer** erfüllt.

Unmittelbar nach einer While oder For-Schleife ist die Bedingung immer falsch.

Frage: Was ist eine geeignete Invariante für diese Schleife?

Eingabe per Dialogfenster

Benutzereingaben können über ein Dialogfenster getätigt werden:

Der Aufruf `JOptionPane.showInputDialog(message)` öffnet ein Dialogfenster und liefert den eingegebenen Text als String zurück, Mit `Integer.parseInt` etc. kann man diesen umrechnen.



Vorher mit `import javax.swing.JOptionPane;` importieren.

Eingabe über die Konsole

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
public class Test {
    public static void main(String[] args) throws IOException
    {
        BufferedReader console = new BufferedReader(
            new InputStreamReader(System.in));
        System.out.println("Geben Sie den Betrag ein:");
        double betrag = Double.parseDouble(console.readLine());
        System.out.println("Ok, Sie bekommen " + betrag + " EUR.");
    }
}
```

Wirkung

```
mhofmann@branford:~/work/teaching/InfoII> java Test
```

```
Geben Sie den Betrag ein:
```

```
45.9
```

```
Ok, Sie bekommen 45.9 EUR.
```

```
mhofmann@branford:~/work/teaching/InfoII>
```

Ausnahmen

Wie in OCAML können während der Auswertung *Ausnahmen* (exceptions) ausgelöst (“geworfen”) werden.

Später lernen wir, wie diese aufgefangen werden können.

Für den Moment führt eine Ausnahme zum Programmabbruch mit Fehlermeldung aus der Klasse und Zeilennummer des Auftretens hervor.

Bestimmte Ausnahmen, wie `IOException` müssen mit `throws` deklariert werden, falls die Möglichkeit ihrer Auslösung besteht.

Einen String in Tokens zerlegen

Ein String enthalte einen Double, einen Integer und einen String, durch Leerzeichen getrennt. Wie kann man diese Daten herauslesen ?

Beispiel: "5 7.0E12 Johanna" besteht aus drei Tokens: "5", "7.0E12" und "Johanna".

Die Klasse `StringTokenizer` gestattet es, einen String in seine "Tokens" zu zerlegen. Ein Token ist ein Teilstring, der kein Leerzeichen enthält.

StringTokenizer

```
import java.util.StringTokenizer;

/* s der zu zerlegende String */
StringTokenizer tokenizer = new StringTokenizer(s);

while (tokenizer.hasMoreTokens()) {
    String token = tokenizer.nextToken();
    /* hier kann man token verarbeiten */
}
```

Aufgabe

Wir wollen alle eingegebenen Wörter der Reihe nach ausgeben, jedes in einer extra Zeile.

Dazu lesen wir die Wörter zeilenweise ein und zerlegen jede Zeile in Tokens, die wir jeweils ausgeben.

Lösung

```
import java.io.*;import java.util.*;
public class Woerter {
    public static void main(String[] args) throws IOException{
        BufferedReader konsole = new BufferedReader(
            new InputStreamReader(System.in));
        boolean fertig = false;
        String line;
        while ( !fertig ) {
            line = konsole.readLine();
            fertig = (line == null);
            if (!fertig) {
                StringTokenizer tokenizer = new StringTokenizer(li
                while (tokenizer.hasMoreTokens()) {
                    System.out.println(tokenizer.nextToken());
                }
            }
        }
    }
}
```

Umlenkung von Ein-/Ausgabe

Man kann die Ausgabe eines Programms in eine Datei umlenken und die Eingabe von einer Datei nehmen:

```
java Woerter < eingabe.in > ausgabe.out
```

liest statt von der Tastatur aus der Datei `eingabe.in` und schreibt statt auf den Bildschirm auf `ausgabe.out`.

```
java Woerter < eingabe.in
```

geht auch und ebenso

```
java Woerter > ausgabe.out
```

ja sogar

```
java Woerter < eingabe.in | sort | java Unique > ausgabe.out
```

wenn etwa `java Unique` **aufeinanderfolgende** Dubletten entfernt.

Mit `|` (pipe) lenkt man die Ausgabe eines Programms direkt in ein anderes Programm als Eingabe um.

Geschachtelte Schleifen

Im Rumpf einer Schleife kann wieder eine solche stehen.

Anwendungsbeispiel: Ausgabe einer Tabelle der Potenzen x^y für $x = 1..10$,
 $y = 1..8$.

1	1	1	1	1	1	1	1	1
2	4	8	16	32	64	128	256	512
3	9	27	81	243	729	2187	6561	19683
4	16	64	256	1024	4096	16384	65536	262144
5	25	125	625	3125	15625	78125	390625	1953125
6	36	216	1296	7776	46656	279936	1679616	10077696
7	49	343	2401	16807	117649	823543	5781344	40353607
8	64	512	4096	32768	262144	2097152	16777216	134218784
9	81	729	6561	59049	531441	4782969	43046721	387420481
10	100	1000	10000	100000	1000000	10000000	100000000	1000000000

Lösung

```
public class Powers{
    public static void main(String[] args){
        final int COLUMN_WIDTH = 10;
        for (int x = 1; x <= 10; x++) {
            for (int y = 1; y <= 8; y++) {
                int p = Math.round(Math.pow(x,y));
                String pstr = "" + p;
                while(pstr.length() < COLUMN_WIDTH)
                    pstr = " " + pstr;
                System.out.print(pstr);

            }
            System.out.println();
        }
    }
}
```

Computersimulation

Auf einem Papier befinden sich parallele Linien im Abstand 2cm.

Eine Nadel der Länge 1cm wird zufällig auf das Papier geworfen.

Wie wahrscheinlich ist es, dass eine Linie getroffen wird?

Das untere Ende der Nadel liege auf Höhe $0 \leq y_{\text{low}} \leq 2$ (bezogen auf die Linie unmittelbar unterhalb der Nadel)

Der Winkel der Nadel betrage $0 \leq \alpha \leq 180$.

Beide Größen (y_{low} und α) seien gleichverteilt.

Das obere Ende der Nadel liegt dann auf Höhe $y_{\text{high}} = y_{\text{low}} + \sin(\alpha)$.

Ein Treffer liegt vor, wenn $y_{\text{high}} \geq 2$.

Näherungsweise Bestimmung der Trefferrate durch Simulation:

Zufallsgenerator

Die Klasse `Random` stellt einen Zufallsgenerator bereit. Die Methode `nextDouble()` liefert einen “zufälligen” `Double`-Wert im Bereich $[0, 1]$

```
import java.util.Random;
public class RandomTest{
    public static void main(String[] args){
        Random gen = new Random();
        System.out.println(" "+ gen.nextDouble() + " "
                           + gen.nextDouble());
    }
}
```

Druckt zwei “Zufallszahlen” aus, z.B.:

```
0.3119991282517587 0.2614453715060384
```

Der Aufruf `gen.nextInt(n)` liefert einen “zufälligen” `Integer` im Bereich $0 \dots n - 1$.

Buffon - Simulation

```
import java.util.Random;
public class Buffon
{ public static void main(String[] args) {
    Random generator = new Random();
    int hits = 0;
    final int NTRIES = 100000000;
    for (int i = 1; i <= NTRIES; i++) {
        double ylow = 2 * generator.nextDouble();
        double angle = 180. * generator.nextDouble();
        double yhigh = ylow + Math.sin(Math.toRadians(angle));
        if (yhigh >= 2) hits++;
    }
    System.out.println("Tries / hits:" + (NTRIES * 1.0) / hits);
}}
```

Ergebnis

... dauert ein paar Minuten und ist

```
Tries / hits:3.1414311574995777
```

Analytische Lösung

$$\begin{aligned} & \Pr(y_{\text{low}} + \sin(\alpha) \geq 2) \\ = & \frac{1}{2} \int_{y=1}^2 \Pr(\sin(\alpha) \geq 2 - y) \, dy \\ = & \frac{1}{2} \int_{y=1}^2 1 - \Pr(\sin(\alpha) \leq 2 - y) \, dy \\ = & \frac{1}{2} \int_{y=1}^2 1 - 2 \cdot \Pr(\alpha \leq \arcsin(2 - y)) \, dy \\ = & \frac{1}{2} \int_{y=1}^2 1 - 2 \cdot \arcsin(2 - y) / \pi \, dy \end{aligned}$$

Ausrechnen des Integrals

```
mhofmann@linux:~/work/teaching/EinfInfo> maple
  |\^/|      Maple 7 (IBM INTEL LINUX)
._|\|      |/|_ . Copyright (c) 2001 by Waterloo Maple Inc.
 \  MAPLE  / All rights reserved. Maple is a registered tra
<_____>   Waterloo Maple Inc.
      |      Type ? for help.
> 1/2 * int(1 - 2 * arcsin(2 - t) / Pi, t=1..2);
                                     1
                                     ----
                                     Pi
```

Zusammenfassung: Schleifen

- Schleifen dienen zur wiederholten Ausführung von Statements.
- Es gibt die `while`-Schleife und die `for`-Schleife. Die `for`-Schleife wird benutzt, wenn im Verlauf der Schleife ein numerischer Wert in konstanten Schritten herauf- oder heruntergezählt wird.
- Invarianten dienen dazu, sich von der Korrektheit einer Schleife zu überzeugen. Hoare Logik formalisiert diese Methode.
- Die Konsolenein- und -ausgabe kann auf Dateien umgelenkt werden.
- Mit Zufallszahlen innerhalb einer Schleife lassen sich Zufallsexperimente simulieren.