Einführung in die Programmierung mit Java

Teil 4: Iteration

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

3. November 2015

Inhalt Teil 4: Iteration

Die While-Schleife

BNF-Beschreibung:

$$\begin{array}{rcl} \langle \textit{statement} \rangle & ::= & \ldots \\ & | & \texttt{while(}\langle \textit{expression} \rangle\texttt{)}\langle \textit{statement} \rangle \\ & & \ldots \end{array}$$

Ausführung von `while(`e`)`c:
Das Statement c wird solange immer wieder ausgeführt, bis der Ausdruck e den Wert `false` hat.

- Der Ausdruck e muss vom Typ `boolean` sein.
- Der Ausdruck e wird vor jeder Ausführung von c ausgewertet. Ist er schon zu Beginn `false`, so wird c überhaupt nicht ausgeführt.
- Bleibt er immer `true`, so wird c immer wieder ausgeführt. Das Programm befindet sich dann in einer **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.

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");
    }
}
```

Schnelles Potenzieren

Folgendes Programmstück berechnet die Potenz a^n uns speichert das Ergebnis in Variable `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 3^5

```
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; }
```

nach Schleifendurchlauf	r	b	i
0	1	3	5
1	3	9	2
2	3	81	1
3	243	6561	0

Für höhere Exponenten werden weniger Multiplikationen benötigt:
Berechnung a^{32} braucht anstatt 32 nur noch 6 Multiplikationen.
Diese Berechnungsmethode ist also “schneller”!

Begründung

Wie funktioniert das ?

Es basiert auf den mathematischen Gleichungen

$$x^{2i+1} = x^{2i}x \qquad \text{und} \qquad x^{2i} = (x^i)^2$$

Wie kann man formal beweisen, dass der Algorithmus immer korrekt funktioniert?

Invariante

Beweis (1/3)

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

Die Eigenschaft I gilt trivialerweise zu Beginn der Schleife, denn dort gilt $\mathtt{r} = 1$, $\mathtt{b} = a$, $\mathtt{n} = i$.

Gilt sie vor Ausführung des Schleifenrumpfes, so gilt sie auch danach. Beweis dieses Teils auf folgenden Folien

Somit gilt sie auch bei Verlassen der Schleife.

Da dann $\mathtt{i} = 0$ gilt, folgt $\mathtt{r} = \mathtt{a}^{\mathtt{n}}$ wie benötigt.

Beweis (2/3)

Es bezeichne $\mathtt{r}_{\text{alt}}, \mathtt{r}_{\text{neu}}$ den Wert von `r` vor und nach dem Schleifenrumpf. Ebenso für Variablen `b` und `i`.

Wir unterscheiden nun zwei Fälle, und betrachten diese einzeln.

Falls $\mathtt{i}_{\text{alt}}$ gerade ist, so gilt dem Code entsprechend:

$$\begin{aligned} \mathtt{b}_{\text{neu}} &= \mathtt{b}_{\text{alt}}^2 \\ \mathtt{i}_{\text{neu}} &= \mathtt{i}_{\text{alt}}/2 \\ \mathtt{r}_{\text{neu}} &= \mathtt{r}_{\text{alt}} \end{aligned}$$

Somit gilt in diesem Fall $\mathtt{r}_{\text{neu}}\mathtt{b}_{\text{neu}}^{\mathtt{i}_{\text{neu}}} = \mathtt{r}_{\text{alt}}\mathtt{b}_{\text{alt}}^{\mathtt{i}_{\text{alt}}}$ wie benötigt.

Beweis (3/3)

Falls $\mathtt{i}_{\text{alt}}$ ungerade ist, so gilt:

$$\begin{aligned} \mathtt{b}_{\text{neu}} &= \mathtt{b}_{\text{alt}}^2 \\ \mathtt{i}_{\text{neu}} &= (\mathtt{i}_{\text{alt}} - 1)/2 \\ \mathtt{r}_{\text{neu}} &= \mathtt{r}_{\text{alt}} \cdot \mathtt{b}_{\text{alt}} \end{aligned}$$

Wir rechnen:
$\mathtt{r}_{\text{neu}}\mathtt{b}_{\text{neu}}^{\mathtt{i}_{\text{neu}}} = \mathtt{r}_{\text{alt}} \cdot \mathtt{b}_{\text{alt}} \cdot \mathtt{b}_{\text{alt}}^{2\mathtt{i}_{\text{neu}}} = \mathtt{r}_{\text{alt}} \cdot \mathtt{b}_{\text{alt}} \cdot \mathtt{b}_{\text{alt}}^{\mathtt{i}_{\text{alt}}-1} = \mathtt{r}_{\text{alt}}\mathtt{b}_{\text{alt}}^{\mathtt{i}_{\text{alt}}}$.

Damit haben wir nun auch die Behauptung am Anfang des Beweises gezeigt. □

Die Eigenschaft *I* heißt **Invariante**.
Sie gilt unverändert vor und nach jedem Schleifendurchlauf.

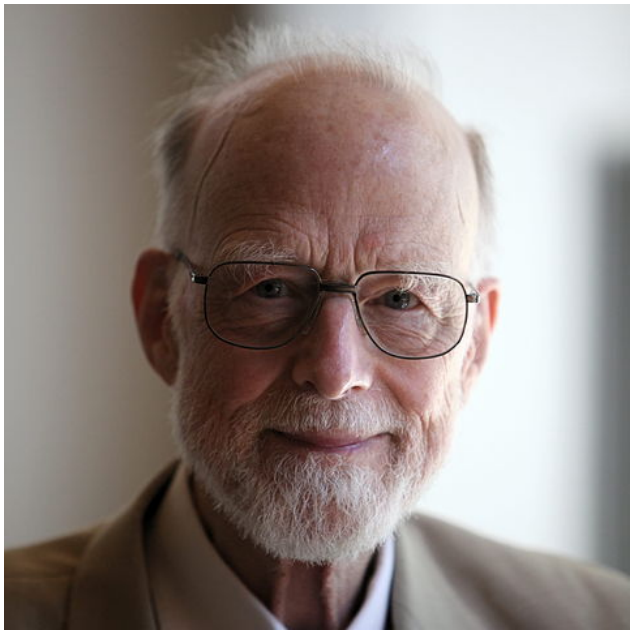
Es bietet sich oft an, bei Schleifen nach geeigneten Invarianten zu suchen.

Hoare Logik

Um die Korrektheit von imperativer Software formell zu beweisen, entwickelte der britische Informatiker C.A.R. Hoare um 1969 zur Systematisierung die Hoare Logik.

Dies ist ein Menge von logischen Regeln, die wir direkt an einem gegebenen, imperativen Quellcode anwenden können, um mathematische Aussagen darüber zu treffen.

Sir C.A.R. "Tony" Hoare (*1934) ist unter Anderem noch bekannt für den Quicksort-Algorithmus und den Null-Pointer.



C.A.R. Hoare

Quelle: Wikimedia Commons
Rama, Cc-by-sa-2.0-fr

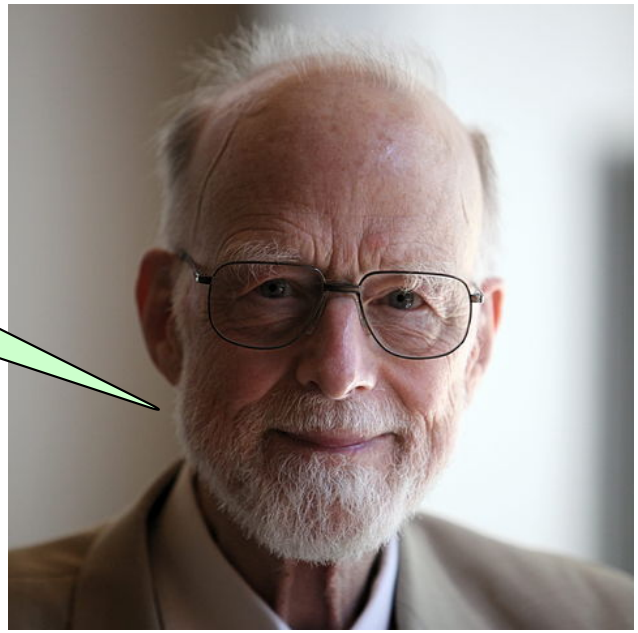
Hoare Logik

ftware formell zu beweisen,
.A.R. Hoare um 1969

> *"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."*
> *during 1980 ACM Turing Award Lecture*

Aussagen darüber zu treffen.

Sir C.A.R. "Tony" Hoare (*1934) ist unter Anderem noch bekannt für den Quicksort-Algorithmus und den Null-Pointer.



C.A.R. Hoare

Quelle: Wikimedia Commons
Rama, Cc-by-sa-2.0-fr

Hoare Logik

Ein **Hoare-Tripel** ist ein formaler Ausdruck der Form

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

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

Definition

Ein solches Hoare-Tripel ist **gültig**, wenn für jeden Programmzustand q, der die **Vorbedingung** P erfüllt, gilt: Falls die Abarbeitung von c ausgehend von Zustand q terminiert mit einem Folgezustand q', so muss dieser Folgezustand q' die **Nachbedingung** Q erfüllen.

P, Q werden auch als **Zusicherungen** bezeichnet.
Wichtig: Für unsere Zwecke sind "Aussagen" deutsche oder englische Sätze, die informelle Mathematik- und Logiknotation verwenden dürfen. Für formale Programmverifikation benötigt man eine formalisierte **Zusicherungssprache**.

Logiknotation

Mathematik	Natürliche Sprache	Java	Warheitstafel
$\neg A$	"nicht A"	!	$\neg$ | T | F F | T
$A \wedge B$	"A und B"	&&	$\wedge$ | T F T | T F F | F F
$A \vee B$	"A oder B"	||	$\vee$ | T F T | T T F | T F
$A \to B$	"A impliziert B"		$\to$ | T F $A = T$ | T F F | T T

Die Aussage $A \to B$ ist äquivalent zu $\neg A \vee B$

Hoare'sche Regeln

Es gibt nun eine Menge von (syntaxgerichteten) Regeln, die es gestatten, die gültigen Hoare-Tripel formal herzuleiten.

Die Regeln haben immer die Form:

$$\frac{P_1 \quad \cdots \quad P_n}{K}$$

Dabei sind $P_1, \ldots, P_n$ die **Prämissen** und K die **Konklusion** der Regel. Meist sind dies Hoare-Tripel oder andere "Aussagen".
Die Reihenfolge der Prämissen ist dabei unerheblich.

Wenn wir die Gültigkeit aller Prämissen zeigen können, dann gibt uns die Anwendung einer passenden Hoare-Regel die Gültigkeit der Konklusion.

Regel für While-Schleifen

$$\frac{P \to I \qquad \{I \wedge b\}c\{I\} \qquad I \wedge \neg b \to Q}{\{P\}\texttt{while(}b\texttt{)}c\{Q\}}$$

Um zu zeigen, dass $\{P\}\texttt{while(}b\texttt{)}c\{Q\}$ gültig ist, muss eine geeignete Aussage 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.

Zur Vereinfachung setzen wir 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.

Beispiel

$$\{\texttt{x} = c \wedge \texttt{y} = d\}\ \texttt{while(x>0)\{y=y+1; x=x-1;\}}\ \{y \geq c + d\}$$

Beispiel

$$\frac{\{I \wedge \texttt{x} > 0\}\,\texttt{\{y=y+1; x=x-1;\}}\,\{I\} \qquad \texttt{x} = c \wedge \texttt{y} = d \;\rightarrow\; I \qquad I \wedge \texttt{x} \leq 0 \;\rightarrow\; y \geq c + d}{\{\texttt{x} = c \wedge \texttt{y} = d\}\,\texttt{while(x>0)\{y=y+1; x=x-1;\}}\,\{y \geq c + d\}}$$

Welche Invariante?

Beispiel

$$\frac{\{I \wedge \texttt{x} > 0\}\,\texttt{\{y=y+1; x=x-1;\}}\,\{I\} \qquad \texttt{x} = c \wedge \texttt{y} = d \;\rightarrow\; I \qquad I \wedge \texttt{x} \leq 0 \;\rightarrow\; y \geq c + d}{\{\texttt{x} = c \wedge \texttt{y} = d\}\,\texttt{while(x>0)\{y=y+1; x=x-1;\}}\,\{y \geq c + d\}}$$

Welche Invariante?

Einsetzen von $\texttt{x} + \texttt{y} = c + d$ für Invariante I erfüllt alle hier geforderten Aussagen:

$$\begin{aligned} (\texttt{x} = c \wedge \texttt{y} = d) &\;\rightarrow\; (\texttt{x} + \texttt{y} = c + d) \\ (\texttt{x} + \texttt{y} = c + d) \wedge \texttt{x} \leq 0 &\;\rightarrow\; y \geq c + d \end{aligned}$$

Beispiel

$$\frac{\{I \wedge \texttt{x} > 0\}\,\texttt{\{y=y+1; x=x-1;\}}\,\{I\} \qquad \begin{array}{c} \texttt{x} = c \wedge \texttt{y} = d \;\rightarrow\; I \\ I \wedge \texttt{x} \leq 0 \;\rightarrow\; y \geq c + d \end{array}}{\{\texttt{x} = c \wedge \texttt{y} = d\}\,\texttt{while(x>0)\{y=y+1; x=x-1;\}}\,\{y \geq c + d\}}$$

Welche Invariante?

Einsetzen von $\texttt{x} + \texttt{y} = c + d$ für Invariante I erfüllt alle hier geforderten Aussagen:

$$\begin{array}{rcl} (\texttt{x} = c \wedge \texttt{y} = d) & \rightarrow & (\texttt{x} + \texttt{y} = c + d) \\ (\texttt{x} + \texttt{y} = c + d) \wedge \texttt{x} \leq 0 & \rightarrow & y \geq c + d \end{array}$$

Es verbleibt zu zeigen:

$\{\texttt{x} + \texttt{y} = c + d \wedge \texttt{x} > 0\}\,\texttt{\{y=y+1; x=x-1;\}}\,\{\texttt{x} + \texttt{y} = c + d\}$

Regel für die Hintereinanderausführung

$$\frac{\{P\}c_1\{R\} \qquad \{R\}c_2\{Q\}}{\{P\}c_1\,;c_2\{Q\}}$$

Hier bezeichnet $c_1\,;c_2$ die Hintereinanderausführung von c_1, c_2.

Für Java müsste man eigentlich $\{c_1 c_2\}$ anstatt $c_1\,;c_2$ schreiben, aber zu viele geschweifte Klammern verwirren dann.
Auch die Verallgemeinerung auf mehr als zwei aufeinanderfolgende Statements belassen wir implizit: Für den Block $\{c_1 c_2 \cdots c_n\}$ mit Vorbedingung P und Nachbedingung Q benötigen wir Zusicherungen R_i mit $R_0 \equiv P$ und $R_n \equiv Q$ und die Hoare-Tripel $\{R_{i-1}\}\, c_i\, \{R_i\}$.

Es hat den Anschein, als müsste man auch hier 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.

Beispiel

Falls c_2 eine Zuweisung ist, so erhält man R einfach durch rückwärts gerichtetes einsetzen der Zuweisung in Q.

Beispiel (Fortsetzung)

Es verbleibte zu zeigen:

$$\{x + y = c + d \wedge x > 0\}\,\texttt{\{y=y+1; x=x-1;\}}\,\{x + y = c + d\}$$

Anwendung der Regel:

$$\frac{\{x + y = c + d \wedge x > 0\}\,\texttt{y=y+1;}\,\{R\} \qquad \{R\}\,\texttt{x=x-1;}\,\{x + y = c + d\}}{\{x + y = c + d \wedge x > 0\}\,\texttt{\{y=y+1; x=x-1;\}}\,\{x + y = c + d\}}$$

Beispiel (Fortsetzung)

Es verbleibte zu zeigen:

$$\{\texttt{x} + \texttt{y} = c + d \wedge \texttt{x} > 0\}\,\texttt{\{y=y+1; x=x-1;\}}\,\{\texttt{x} + \texttt{y} = c + d\}$$

Anwendung der Regel:

$$\frac{\{\texttt{x} + \texttt{y} = c + d \wedge \texttt{x} > 0\}\,\texttt{y=y+1;}\,\{R\} \qquad \{R\}\,\texttt{x=x-1;}\,\{\texttt{x} + \texttt{y} = c + d\}}{\{\texttt{x} + \texttt{y} = c + d \wedge \texttt{x} > 0\}\,\texttt{\{y=y+1; x=x-1;\}}\,\{\texttt{x} + \texttt{y} = c + d\}}$$

Für R setzen wir hier $x + y = c + d + 1$ ein, womit es verbleibt, noch die Gültigkeit der folgenden beiden Hoare-Tripel zu zeigen:

$$\{\texttt{x} + \texttt{y} = c + d \wedge \texttt{x} > 0\}\quad \texttt{y=y+1;}\quad \{\texttt{x} + \texttt{y} = c + d + 1\}$$
$$\{\texttt{x} + \texttt{y} = c + d + 1\}\quad \texttt{x=x-1;}\quad \{\texttt{x} + \texttt{y} = c + d\}$$

Regel für die Zuweisung

Sei e ein seiteneffektfreier Ausdruck und x eine Programmvariable. Für das Zuweisungsstatement x=e; gilt die folgende Hoare Regel:

$$\frac{P \to Q[x := e]}{\{P\}x\texttt{=}e\texttt{;}\{Q\}}$$

Bedeutung von $Q[x := e]$

Für jeden Zustand q ist die Zusicherung Q genau dann erfüllt, wenn $Q[x := e]$ in einem Zustand q' erfüllt ist, wobei sich q' von q nur dadurch unterscheidet, dass x den Wert e in q' hat.

Beispiel

Ist $Q \equiv$ "$x = 19$", also in all den Zuständen erfüllt, in denen x den Wert 19 hat, so ist $Q[x := x\texttt{+}1] \equiv$ "$x + 1 = 19$", also "$x = 18$".
Es gilt also z.B.: $\{x = 18\}x\texttt{=}x\texttt{+}1\texttt{;}\{x = 19\}$.

Beispiel (Fortsetzung)

Es verbleibte zu zeigen:

$$\{\texttt{x} + \texttt{y} = c + d \wedge \texttt{x} > 0\} \quad \texttt{y=y+1;} \quad \{\texttt{x} + \texttt{y} = c + d + 1\}$$
$$\{\texttt{x} + \texttt{y} = c + d + 1\} \quad \texttt{x=x-1;} \quad \{\texttt{x} + \texttt{y} = c + d\}$$

Anwendung der Regeln:

Wenn $x + y = c + d \wedge x > 0$ gilt, dann gilt auch $x + (y + 1) = c + d + 1$. Die Anwendung der Zuweisung erlaubt uns dann, $(y + 1)$ durch y zu ersetzen.

Wenn $x + y = c + d + 1$ gilt, dann gilt auch $(x - 1) + y = c + d$. Die Anwendung der Zuweisung erlaubt uns dann, $(x - 1)$ durch x zu ersetzen wie benötigt.

Beispiel (Zusammenfassung)

$$
\begin{array}{rl}
\{\mathtt{x} = c \wedge \mathtt{y} = d\} & \\
\{\mathtt{x} + \mathtt{y} = c + d\} & \texttt{while(x>0)\{} \\
\{\mathtt{x} > 0 \wedge \mathtt{x} + \mathtt{y} = c + d\} & \\
\{\mathtt{x} + (\mathtt{y+1}) = c + d + 1\} & \texttt{\quad y=y+1;} \\
\{\mathtt{x} + \mathtt{y} = c + d + 1\} & \\
\{(\mathtt{x-1}) + \mathtt{y} = c + d\} & \texttt{\quad x=x-1;} \\
\{\mathtt{x} + \mathtt{y} = c + d\} & \texttt{\}} \\
\{\mathtt{x} \leq 0 \wedge \mathtt{x} + \mathtt{y} = c + d\} & \\
\{y \geq c + d\} &
\end{array}
$$

Bemerkung:

Es bietet sich oft an, solche Beweise rückwärts aufzubauen, so dass man immer mit der schwächsten Nachbedingung arbeitet, welche gerade noch stark genug ist.

Konsequenz-Regel

Im behandelten Beispiel haben wir z.B. innerhalb der Schleife von $x > 0$ keinen Gebrauch gemacht.

Generell ist erlaubt:

- schwächere Vorbedingung zu fordern
- stärkere Nachbedingungen zu garantieren

Formalierung als Regel:

$$\frac{P \to P' \qquad \{P'\}c\{Q'\} \qquad Q' \to Q}{\{P\}c\{Q\}}$$

Regeln für die Fallunterscheidung

Sei b ein seiteneffektfreier Ausdruck vom Typ `boolean`.
Es gelten die folgenden Hoare Regeln:

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

Wir müssen also beide Möglichkeiten einzelen behandeln; dabei bekommen wir als zusätzliche Vorbedingung, dass b entsprechend dem Fall gilt bzw. nicht gilt.

Die Regel hat eine Spezialisierung für den Fall, dass es keinen `else`-Zweig gibt:

$$\frac{\{P \wedge b\}c\{Q\} \qquad P \wedge \neg b \to Q}{\{P\}\texttt{if(}b\texttt{)}\ c\{Q\}}$$

Beispiel

Betrachten wir das folgende Hoare-Tripel:

$$\{x > 5\}\ \texttt{while (x>5) \{ y=y+1; \}}\ \{x \leq 5\}$$

Ist dieses Hoare-Tripel gültig?

Wenn ja, was bedeutet es?

Partielle Korrektheit

Angenommen das Hoare-Tripel $\{P\}\, c\, \{Q\}$ wäre gültig.

Dies bedeutet:

Für jeden Programmzustand, der Prämisse P erfüllt, gilt nach Abarbeitung von c, dass Konklusion Q gilt; *falls* die Abarbeitung terminiert!

Mit den behandelten Regeln läßt sich nur **partielle Korrektheit** beweisen, d.h. ein Programm arbeitet Korrektheit unter der Voraussetzung, dass es terminiert und auch, dass keine Ausnahme geworfen wird.

Totale Korrektheit bedeutet dagegen, dass das Programm immer teminiert und auch das keine Fehler auftreten.

Es gibt Versionen von Hoare Logik, welche auch den Beweis der totalen Korrektheit erlauben; diese behandeln wir hier jedoch nicht.

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 Hoare-Regeln für alle anderen Java Konstrukte, insbesondere Methodenaufrufe, 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.
- Man kann diese Idee so systematisieren, dass eine Herleitung in der Hoare Logik aus geeigneten Invarianten für die Schleifen automatisch erzeugt werden kann.

Übersicht Hoare Logik

$$\frac{P \to I \qquad \{I \wedge b\}c\{I\} \qquad I \wedge \neg b \to Q}{\{P\}\texttt{while(}b\texttt{)}\,c\{Q\}} \quad \text{(While)}$$

$$\frac{\{P\}c_1\{R\} \qquad \{R\}c_2\{Q\}}{\{P\}c_1\texttt{;}\,c_2\{Q\}} \quad \text{(Komposition)}$$

$$\frac{P \to Q[x := e]}{\{P\}x\texttt{=}e\texttt{;}\{Q\}} \quad \text{(Zuweisung)}$$

$$\frac{\{P \wedge b\}c_1\{Q\} \qquad \{P \wedge \neg b\}c_2\{Q\}}{\{P\}\texttt{if(}b\texttt{)}\ c_1\ \texttt{else}\ c_2\{Q\}} \quad \text{(If-Else)}$$

$$\frac{\{P \wedge b\}c\{Q\} \qquad P \wedge \neg b \to Q}{\{P\}\texttt{if(}b\texttt{)}\ c\{Q\}} \quad \text{(If)}$$

$$\frac{P \to P' \qquad \{P'\}c\{Q'\} \qquad Q' \to Q}{\{P\}c\{Q\}} \quad \text{(Konsequenz)}$$

Beispiel: Hoare Logik

In einer Vergangenen EiP-Klasur wurde folgende Aufgabe gestellt:
Geben ist folgendes Programmfragment c:

```
r = z;
while (r >= n) {
  q = q + 1;
  r = r - n;
}
```

Beweisen Sie die Gültigkeit des Hoare-Triple

$$\{q = 0 \land n > 0\}\ c\ \{z = q \cdot n + r \land r < n\}$$

1. Was gilt direkt vor Beginn der Schleife?
2. Welche Invariante wählen Sie für die Schleife?
3. Was gilt nach Abarbeitung des Programmfragments?
4. Was gilt direkt vor Abarbeitung des Schleifenrumpfes?
5. Was gilt zwischen den Zuweisungen im Schleifenrumpf?
6. Was gilt direkt nach Abarbeitung des Schleifenrumpfes?

Lösung (1/3)

Zuerst behandeln wir die Zuweisung vor der Schleife, um herauszufinden, was unmittelbar vor der Schleife gilt. Dies ist ein gewöhnliches Hoare-Tripel für Zuweisungen:

$$\{q = 0 \land n > 0\}\ \texttt{r=z;}\ \{q = 0 \land n > 0 \land r = z\}$$

Als Invariante wählen wir dann:

$$z = q \cdot n + r$$

Dies ist letzendlich nur die gegebene Nachbedingung (abzüglich negierter Schleifenbedingung), welche wir ja aus der Invariante zusammen mit der negierten Schleifenbedingung herleiten müssen (gemäß der Hoare-Regel für While-Schleifen).

Im Allgemeinen kann es auch sein, dass die Invariante echt stärker ist als die geforderte Nachbedingung (z.B. zusätzliche Annahmen, welche benötigt werden um die Invariante nach Durchlauf des Schleifenrumpfes wieder herzustellen.

Lösung (2/3)

Die Invariante gilt trivialerweise vor Beginn der Schleife:

$$z = 0 \cdot n + r$$

Die Invariante zusammen mit der negierten Schleifenbedingung ist identisch zur geforderten Nachbedingung

$$z = q \cdot n + r \wedge r < n$$

denn $\neg(r \geq n) \equiv r < n$.

Zur Anwendung der Hoare-Regel für While-Schleifen müssen wir also nur noch die Gültigkeit des folgenden Hoare-Tripels beweisen:

$$\{q \cdot n + r \wedge r \geq n\}\ \texttt{q=q+1;r=r-n}\ \{q \cdot n + r\}$$

Lösung (3/3)

$$\begin{array}{rl}
 & \{q \cdot n + r \ \wedge \ r \geq n\} \\
\{(\underbrace{q+1}_{q:=} - 1) \cdot n + r \ \wedge \ r \geq n\} & \texttt{q=q+1;} \\
\{(q-1) \cdot n + r \ \wedge \ r \geq n\} & \\
\{q \cdot n + \underbrace{r-n}_{r:=} \ \wedge \ r \geq n\} & \texttt{r=r-n;} \\
\{q \cdot n + r \ \wedge \ r \geq n\} & \\
\{q \cdot n + r\} &
\end{array}$$

Zeilen ohne Code sind Anwendungen der Konsequenz-Regel;
Zeilen mit Code sind Anwendung der Zuweisungs-Regel;
der gesamte Block folgt aus der Kompositions-Regel.
(Nachbedinungen auf rechter Seite weggelassen, da identisch zur Vorbedingung in der Zeile darunter)

Der Beweis wurde hier von oben-nach-unten konstruiert. Sonst hätte man das unnötige $r \geq n$ wohl gleich ganz oben weggworfen.

Macht aber ja keinen großen Unterschied.

Stärkste vs. Schwächste Zusicherung

Was bedeutet die Gültigkeit folgender Hoare-Tripel:

1. $\{\text{true}\}\ c\ \{Q\}$
2. $\{\text{false}\}\ c\ \{Q\}$
3. $\{P\}\ c\ \{\text{true}\}$
4. $\{P\}\ c\ \{\text{false}\}$

Stärkste vs. Schwächste Zusicherung

Was bedeutet die Gültigkeit folgender Hoare-Tripel:

1. $\{\text{true}\}\ c\ \{Q\}$ "Wenn c terminiert, dann gilt danach Q."
   Dies gilt für jeden beliebigen Anfangszustand!
2. $\{\text{false}\}\ c\ \{Q\}$
3. $\{P\}\ c\ \{\text{true}\}$
4. $\{P\}\ c\ \{\text{false}\}$

Stärkste vs. Schwächste Zusicherung

Was bedeutet die Gültigkeit folgender Hoare-Tripel:

1. $\{\text{true}\}\ c\ \{Q\}$ "Wenn c terminiert, dann gilt danach Q."
   Dies gilt für jeden beliebigen Anfangszustand!

2. $\{\text{false}\}\ c\ \{Q\}$ Dieses Tripel ist trivialerweise stets erfüllt: Es gibt ja keinen Zustand, der die Vorbedingung erfüllt, also muss auch die Nachbedingung nicht gelten (sie kann aber gelten).

3. $\{P\}\ c\ \{\text{true}\}$

4. $\{P\}\ c\ \{\text{false}\}$

Stärkste vs. Schwächste Zusicherung

Was bedeutet die Gültigkeit folgender Hoare-Tripel:

1. $\{\mathrm{true}\}\ c\ \{Q\}$ "Wenn c terminiert, dann gilt danach Q."
   Dies gilt für jeden beliebigen Anfangszustand!

2. $\{\mathrm{false}\}\ c\ \{Q\}$ Dieses Tripel ist trivialerweise stets erfüllt: Es gibt ja keinen Zustand, der die Vorbedingung erfüllt, also muss auch die Nachbedingung nicht gelten (sie kann aber gelten).

3. $\{P\}\ c\ \{\mathrm{true}\}$ Auch dieses Tripel ist trivialerweise stets erfüllt, denn true ist die schwächste aller Nachbedingung.

4. $\{P\}\ c\ \{\mathrm{false}\}$

Stärkste vs. Schwächste Zusicherung

Was bedeutet die Gültigkeit folgender Hoare-Tripel:

1. $\{\mathrm{true}\}\ c\ \{Q\}$ "Wenn c terminiert, dann gilt danach Q."
   Dies gilt für jeden beliebigen Anfangszustand!

2. $\{\mathrm{false}\}\ c\ \{Q\}$ Dieses Tripel ist trivialerweise stets erfüllt: Es gibt ja keinen Zustand, der die Vorbedingung erfüllt, also muss auch die Nachbedingung nicht gelten (sie kann aber gelten).

3. $\{P\}\ c\ \{\mathrm{true}\}$ Auch dieses Tripel ist trivialerweise stets erfüllt, denn true ist die schwächste aller Nachbedingung.

4. $\{P\}\ c\ \{\mathrm{false}\}$ "Wenn P gilt, dann terminiert c nicht."
   Nachbedingung false gilt ja per Definition nie. Für jeden Zustand, der P erfüllt, kann c also nicht terminieren, falls das Tripel wirklich beweisbar ist.

Ausdrücke als Statement

BNF

$$\langle \textit{statement} \rangle ::= \ldots \mid \langle \textit{expression} \rangle\,;$$

Bedeutung

Ist e ein beliebiger Ausdruck, so ist also $e;$ 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 $x{=}e$ auch ein Ausdruck ist: Seine Auswertung weist als Seiteneffekt den Wert von e der Variablen x zu. Wert dieses Ausdrucks ist der Wert von e.

Zuweisungen als Ausdruck

Das Zuweisungen eigentlich Ausdrücke mit Seiteneffekt sind, *könnte* man so ausnutzen:

```
    int n = 0;
    while (10 > (n=n+1)) {
      System.out.print(" n="+n);
    }
//Ausgabe: " n=1 n=2 n=3 n=4 n=5 n=6 n=7 n=8 n=9"
```

- Ein Beispiel für Bedingung mit Seiteneffekt! Die behandelten Hoare-Regeln können hier also nicht angewendet werden!
- Nicht besonders leserlich. Besser eigene Zuweisungen:

```
int n = 0; n=n+1;
while (10 > n) { System.out.print(" n="+n); n=n+1; }
```

Andererseits ist schon sehr übersichtlich, alle Information über die Schleife (Start, Bedingung, Schritt) am Anfang der Schleife gesammelt zu haben $\Rightarrow$ `for`-Schleife.

Zuweisungen als Ausdruck

Das Zuweisungen eigentlich Ausdrücke mit Seiteneffekt sind, *könnte* man so ausnutzen:

```
    int n = 0;
    while (10 > (n=(n=n+1)+1)) {
      System.out.print(" n="+n+);
    }
//Ausgabe: " n=2 n=4 n=6 n=8"
```

- Ein Beispiel für Bedingung mit Seiteneffekt! Die behandelten Hoare-Regeln können hier also nicht angewendet werden!
- Nicht besonders leserlich. Besser eigene Zuweisungen:

```
int n = 0; n=n+1;
while (10 > n) { System.out.print(" n="+n); n=n+1; }
```

Andererseits ist schon sehr übersichtlich, alle Information über die Schleife (Start, Bedingung, Schritt) am Anfang der Schleife gesammelt zu haben $\Rightarrow$ `for`-Schleife.

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;
}
```

Nach Ausführung ist `summe` gleich 5050.

Formelle Berschreibung in BNF

⟨*statement*⟩ ::= ...
| `for(`⟨*expression*⟩`;`⟨*expression*⟩`;`⟨*expression*⟩`)`
⟨*statement*⟩

Ausführung von

- `for(`*init*`;`*cond*`;`*step*`)`*body*:
- *init* und *step* sind **Ausdrücke** mit Seiteneffekten (Zuweisungen oder Methodenaufrufe).
- *cond* ist ein Ausdruck vom Typ Boolean.
- *body* ist ein beliebiges Statement.

Dies kann ähnlich zu folgender `while`-Schleife lesen:

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

Formelle Berschreibung in BNF

$$\langle statement \rangle ::= \ldots \mid \texttt{for(}\langle expression \rangle\texttt{;}\langle expression \rangle\texttt{;}\langle expression \rangle\texttt{)}\ \langle statement \rangle$$

Ausführung von

- `for(`*init*`;`*cond*`;`*step*`)body`:
- *init* und *step* sind **Ausdrücke** mit Seiteneffekten (Zuweisungen oder Methodenaufrufe).
- *cond* ist ein Ausdruck vom Typ Boolean.
- *body* ist ein beliebiges Statement.

Dies kann ähnlich zu folgender `while`-Schleife lesen:

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

Unterschied: Die Lebensspanne von Variablen, welche im *init*-Block deklariert sind, endet mit der Schleife!

Beispiel: Zeichenkette umdrehen

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

Man verwendet sie z.B. so:
`"Matthias".length()` ist 8
`"Matthias".charAt(2)` ist `'t'`

Die Methode `charAt` liefert ein Ergebnis vom Typ `char`. Man kann `char`s 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?

Do-While-Schleife

Eine weitere alternative Schleife ist die `do-while`-Schleife. Der einzige Unterschied zur `while`-Schleife ist, dass der Schleifenrumpf *mindestens einmal* durchlaufen wird.
Dies ist manchmal praktisch, um eine Sonderbehandlung am Anfang zu vermeiden.

BNF

$$\langle \textit{statement} \rangle \quad ::= \quad \ldots \;\mid\; \texttt{do}\ \langle \textit{statement} \rangle\ \texttt{while(}\langle \textit{expression} \rangle\texttt{);}$$

Das Statement `do` *rumpf* `;` `while(`*cond*`);` kann ähnlich zu folgender `while`-Schleife lesen:

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

Do-While-Schleife

Eine weitere alternative Schleife ist die `do-while`-Schleife. Der einzige Unterschied zur `while`-Schleife ist, dass der Schleifenrumpf *mindestens einmal* durchlaufen wird.
Dies ist manchmal praktisch, um eine Sonderbehandlung am Anfang zu vermeiden.

BNF

$$\langle \textit{statement} \rangle \quad ::= \quad \ldots \;\mid\; \texttt{do}\ \langle \textit{statement} \rangle\ \texttt{while(}\langle \textit{expression} \rangle\texttt{);}$$

Das Statement `do` *rumpf* `; while(`*cond*`);` kann ähnlich zu folgender `while`-Schleife lesen:

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

Unterschied:
Lebensspanne von Variablen, welche im *rumpf*-Block deklariert sind, gilt sonst nur für *einen* Schleifendurchlauf!

Geschachtelte Schleifen

Im Rumpf einer Schleife darf wieder eine Schleife stehen.

Anwendungsbeispiel:

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

1	1	1	1	1	1	1	1
2	4	8	16	32	64	128	256
3	9	27	81	243	729	2187	6561
4	16	64	256	1024	4096	16384	65536
5	25	125	625	3125	15625	78125	390625
6	36	216	1296	7776	46656	279936	1679616
7	49	343	2401	16807	117649	823543	5764801
8	64	512	4096	32768	262144	2097152	16777216
9	81	729	6561	59049	531441	4782969	43046721
10	100	1000	10000	100000	1000000	10000000	100000000

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 = (int)Math.round(Math.pow(x,y));
                String pstr = "" + p;
                /* Auffuellen bis zur COLUMN_WIDTH */
                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 $y_{\text{high}} \geq 2$.

Idee Bestimmung der Trefferrate durch Simulation!

Zufallsgenerator

Die Klasse `Random` stellt einen Zufallsgenerator bereit. Die Methode `nextDouble()` liefert "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 $0 \ldots n-1$.

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
```

Die Wahrscheinlichkeit beträgt also ungefähr $\frac{1}{\pi}$.

Analytische Lösung

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

Berechnung des Integrals liefert $\frac{1}{\pi} \equiv 0.3183098861837907$

Weitere Beispiele

Was wird hier gedruckt?

```
for (int i = 0; i < 10; i++) {
   for (int j = 0; j < 10; j++)
      System.out.print(i * j % 10);
   System.out.println();}
```

Wie oft werden diese Schleifen ausgeführt?

```
for (i = 1; i <= 10; i++) ...
for (i = -10; i <= 10; i++) ...
for (i= -10; i <= 10; i = i + 3) ...
for (i= -10; i <= 10; i = i - 1) ...
```

Übungsaufgabe: Random Walk

vgl. [Horstmann]
Man simuliere die Wanderung eines "Betrunkenen" in einem "Straßengitter".
Man zeichne ein Gitter von 10 $\times$ 10 Straßen (=Linien) und repräsentiere den "Betrunkenen" als Kreuz in der Mitte des Gitters.
Eine bestimmte Zahl von Malen, z.B. 100, lasse man den Betrunkenen zufällig eine Richtung (N, O, S, W), bewege ihn einen Block weiter in dieser Richtung und zeichne ihn neu.
Anschließend bestimme man die zurückgelegte Entfernung.

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.
- Bevor man eine Schleife programmiert, sollte man an die mögliche Invariante denken!
- Hoare Logik formalisiert die Methode, die Korrektheit eines Programmes zu beweisen.

Iteration zur Abarbeitung von Eingaben

```
import java.util.Scanner;
public class Woerter {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        while (console.hasNextInt()) {
            System.out.println(console.nextInt());
        }
        System.out.print(console.next()+"ist keine Zahl!");
}   }
```

- `Scanner.next()` liefert nächste Eingabe als `String`
- `Scanner.nextInt()` liefert `int`; löst Fehler aus, falls die Eingabe keine ganze Zahl war!
- `Scanner.hasNextInt()` prüft, ob *nächste* Eingabe eine ganze Zahl ist.

Jede dieser Befehl *kann* Eingabeaufforderung auslösen, falls keine Eingabe vorhanden! `Scanner.hasNextInt()` beläßt Eingabe im Scanner – nächster Befehl löst dann keine Eingabeaufforderung aus.

Eingabe über die Konsole

```
public static void main(String[] args) {
  Scanner console = new Scanner(System.in);
  while (console.hasNext()) {
    System.out.println(console.next());
    }
  console.close();
}
```

- Die Methode `next()` liefert das jeweils nächste **Token**
- Ein Token ist ein durch **Leerzeichen** voneinander getrennte Teile der Eingabe Man kann auch andere Symbole als Trennzeichen festlegen.
- Methode `hasNext()` prüft, ob noch Tokens vorhanden sind.
- Methode `nextLine()` liefert komplette Eingabezeile.
- Aufruf `close()` schliesst Eingabe. Nicht vergessen, vor allem bei Datei-Operationen.

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
```

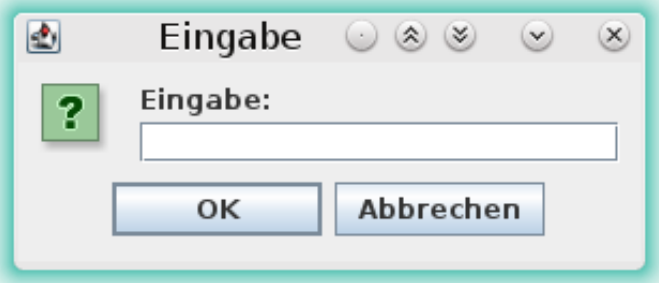
wenn etwa `java Unique` **aufeinanderfolgende** Dubletten entfernt. Mit der Unix-pipe `|` lenkt man die Ausgabe eines Programms direkt in ein anderes Programm als Eingabe um.

Eingabe per Dialogfenster

Alternativ können Benutzereingaben auch hübscher über ein Dialogfenster getätigt werden:

```
import javax.swing.JOptionPane;

public class Main {
  public static void main(String[] args) {
    String input = JOptionPane.showInputDialog("Eingabe: ");
    System.out.println("Input war: "+input);
  }
}
```



Aufruf `JOptionPane.showInputDialog(`*message*`)` öffnet ein Dialogfenster und liefert den eingegebenen Text als String zurück. Vorher mit `import javax.swing.JOptionPane;` importieren.

Zahlen per Dialogfenster

```
import javax.swing.JOptionPane;

public class Main {
  public static void main(String[] args) {
    String msg   = "Bitte Zahl eingeben: ";
    String input = JOptionPane.showInputDialog(msg);
    int x = Integer.parseInt(input);
    System.out.println("Zahl war: "+x);
  }
}
```

Um Zahlen einzulesen, muss man diese mit Methoden wie etwa `Integer.parseInt` umrechnen.

Hier muss man aber immer eine Fehlerbehandlung durchführen, wenn keine Zahl eingegeben wurde.

Fehlerbehandlung kommt aber erst später