

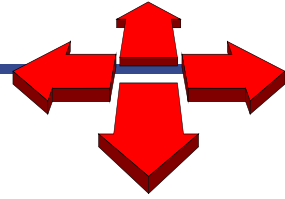
Das SE1-Team

- Die aktuelle Lehrveranstaltung ist das Ergebnis von intensiver Arbeit über mindestens sechs Jahre.
- Sie wäre so nicht möglich ohne den engagierten Einsatz von sehr vielen Personen:
 - Von inzwischen weit über 1000 Studierenden als Teilnehmern;
 - Von zahlreichen Betreuern, sowohl wissenschaftlichen Mitarbeiterinnen und Mitarbeitern als auch studentischen Betreuern;
 - Von vielen (teilweise ehemaligen) Kolleginnen und Kollegen im AB SWT, allen voran momentan Christian Späh.
 - Und nicht zuletzt: von Heinz Züllighoven.

„Glück ist das Produkt einer Kommunikation. Ich kann nicht delegieren, dass jemand mich glücklich macht.“

Renan Demirkan

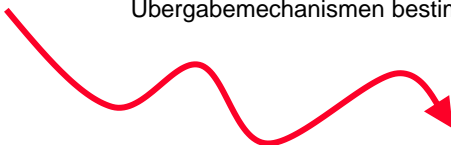
Strukturierte Programmierung



- Die **strukturierte Programmierung** beschränkt sich auf die **Kontrollstrukturen** Sequenz, Verzweigung und Wiederholung und verzichtet auf Sprungbefehle.
- Bei der Verzweigung unterscheiden wir **einfache Verzweigungen** und **Mehrfachverzweigungen**.
- Wiederholungen werden in der klassischen imperativen Programmierung mit Hilfe von **Schleifen-Mechanismen** realisiert.

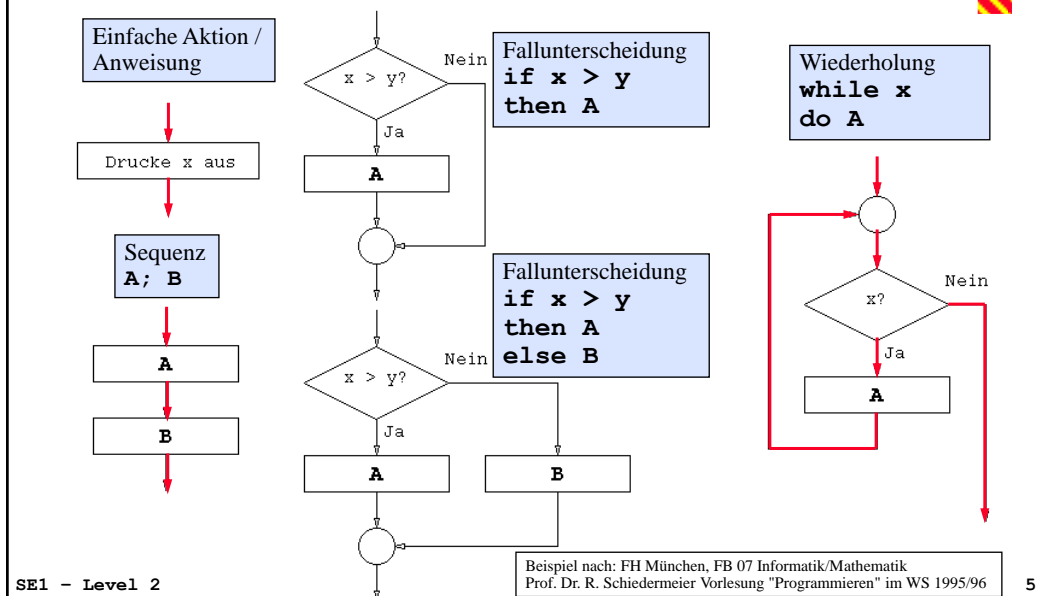
Kontrollfluss

- Das Verständnis des **Kontrollflusses** von (imperativen) Programmiersprachen ist eine wesentliche Voraussetzung für die Entwicklung korrekter Programme. Der Kontrollfluss bestimmt die **Reihenfolge**, in der Teile eines Programms ausgeführt werden.
- Der Kontrollfluss kann auf verschiedenen Ebenen betrachtet werden:
 - **Innerhalb einer Anweisung** (etwa im Ausdruck auf der rechten Seite einer Zuweisung) wird er durch die Bindungsstärke und Assoziativität der Operatoren bestimmt.
 - **Zwischen den Anweisungen** einer Methode wird er durch Kontrollstrukturen bestimmt.
 - **Zwischen den Methoden** wird er durch Methodenaufrufe und Übergabemechanismen bestimmt.

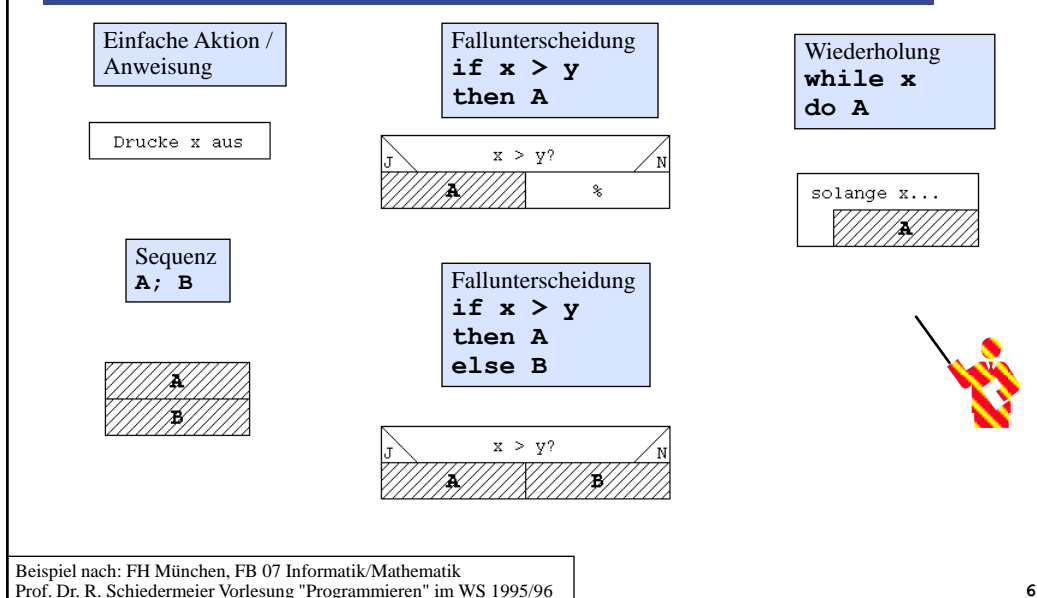


Wir erinnern uns: Wir betrachten in SE1 ausschließlich sequenzielle Abläufe eines Programms, d.h. zu einem Zeitpunkt wird nur jeweils eine Anweisung abgearbeitet.

Darstellungsmittel für Kontrollfluss: Flussdiagramme



Darstellungsmittel für Kontrollfluss: Struktogramme



Die Diskussion um Kontrollstrukturen

- Von Mitte der 60er bis Mitte der 70er wurde in der Softwaretechnik (Informatik) viel über **Kontrollstrukturen** diskutiert.
- Bohm und Jacopini haben 1966 nachgewiesen, dass alle **Algorithmen**, die in **Flußdiagrammen** ausgedrückt werden können, **D-Diagramme** sind und durch entsprechende **Kontrollanweisungen** implementierbar sind.
- Kontrollstrukturen sollen **einen Einstieg** und **einen Ausstieg** (engl.: single entry, single exit) besitzen.

D-Diagramme [Dijkstra]:

- 1 Eine einfache Aktion ist ein D-Diagramm.
- 2 Sind **A** und **B** D-Diagramme, dann sind auch


```
A; B,
if c then A end,
if c then A else B end,
while c do A end
```

 D-Diagramme.
- 3 Nichts sonst ist ein D-Diagramm.

SE1 – Level 2

© Pomberger in Rechenberg, Pomberger

7

„Go To Statement Considered Harmful“

- Obwohl die **unbedingte Verzweigung (Goto)** ausreicht, alle anderen Kontrollstrukturen nachzubilden, führt ihre uneingeschränkte Verwendung zu unlesbaren und unzuverlässigen Programmen.
- **Hauptgrund:**
 - Durch Goto kann im Ablauf jede beliebige Reihenfolge von Anweisungen unabhängig von ihrer textlichen Anordnung erreicht werden.
- In einem berühmten Leserbrief ("**Go To statement considered harmful**", CACM, 1968, Vol.11, No.3, pp.147-148) schreibt E.W. **Dijkstra**:
 - *"The goto statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program."*
- Dies hat die **Goto-Debatte** entzündet, die zwar zur softwaretechnischen Ablehnung des uneingeschränkten Goto geführt hat, aber nur wenige Programmiersprachen haben völlig auf dieses Konstrukt verzichtet.



Java bietet keine Goto-Anweisung; allerdings ist **goto** als Schlüsselwort reserviert...

SE1 – Level 2

© Sebesta

8

Edsger W. Dijkstra zum Go To Statement



“For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all “higher level” programming languages (i.e. everything except, perhaps, plain machine code).”

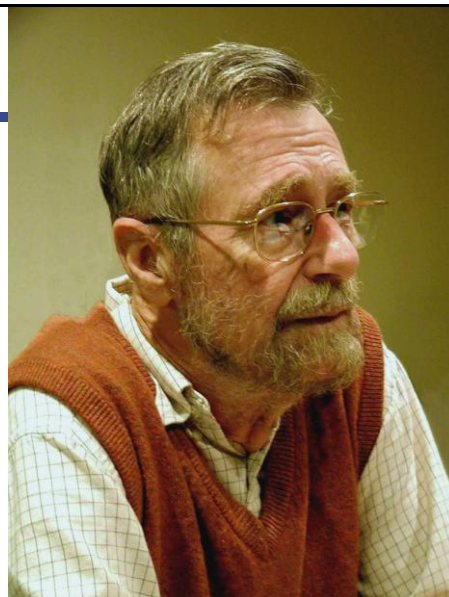
...

Reprinted from Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148. Copyright © 1968, Association for Computing Machinery, Inc.

<http://www.acm.org/classics/oct95/>

Edsger W. Dijkstra

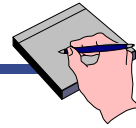
On 6 August 2002, Edsger W. Dijkstra, Professor Emeritus of Computer Sciences and Mathematics at The University of Texas at Austin, died at his home in Nuenen, the Netherlands.



“Separate concerns.”

“Program testing can at best show the presence of errors, but never their absence.”

Strukturierte Programmierung



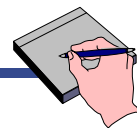
- Die heute klassischen Kontrollstrukturen der imperativen Programmierung sind **Sequenz**, **Auswahl**, **Wiederholung**.
- Sprachen mit diesen Kontrollstrukturen heißen auch **strukturierte Sprachen**, was die enge Beziehung zur **strukturierten Programmierung** verdeutlicht.
- Der Kern der strukturierten Programmierung ist die **Beschränkung** der Kontrollstrukturen in Programmen auf Sequenz, Auswahl und beschränkte Schleifenkonstrukte.
- Dazu kommen die Verwendung von Prozeduren und (abstrakten) Datentypen.



SE1 – Level 2

11

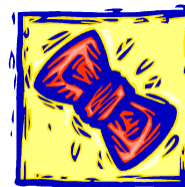
Zusammengesetzte Anweisungen



- **Zusammengesetzte Anweisungen** (engl.: compound statements) fassen eine Folge von Anweisungen zu einer einzigen Anweisung zusammen, indem sie syntaktisch klammern (etwa durch **begin** und **end**).
- Da zusammengesetzte Anweisungen programmiersprachlich als **eine einzige Anweisung** gelten, sind sie in Kontrollstrukturen sehr nützlich.

Zusammengesetzte Anweisung in Algol 60:

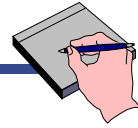
```
begin
  statement_1;
  ...
  statement_n
end
```



SE1 – Level 2

12

Blöcke



- **Blöcke** sind zusammengesetzte Anweisungen, die um **lokale Variablen** ergänzt werden.
- Blöcke bilden einen eigenen Sichtbarkeitsbereich (kommt noch).
- Auch ein Block ist syntaktisch geklammert; in Java mit geschweiften Klammern, in Algol-artigen Sprachen auch durch **begin ... end**
- ALGOL-60 war die erste Sprache mit Blöcken.
- Programmiersprachen, die Blöcke kennen, heißen auch **blockstrukturiert**.

```
{
    int a, b;
    ...
    if (a < b)
    {
        int temp; // Block-lokale Variable
        temp = a;
        a = b;
        b = temp;
    }
}
```



Level 1 Syntax

Block:
 { { BlockStatement } }
 BlockStatement:
 LocalVariableDeclarationStatement
 Statement

SE1 – Level 2

13

Fallunterscheidungen: if-Anweisung

- Die programmiersprachliche Realisierung von **Fallunterscheidungen** heißt auch **Verzweigung** oder **bedingte Anweisung** (engl.: conditional statement).
- Üblich sind **Zweiweg-** und **Mehrweg-Verzweigungen**.
- **Die** Standardform der Zweiweg-Verzweigung ist die **if-Anweisung**. Für Java ist sie syntaktisch folgendermaßen definiert (siehe Java Level 1 Syntax):

Statement:
 if (Expression) Statement [else Statement]

The expression must have type boolean, or a compile-time error occurs.



```
if (a < b)
    min = a;
else
    min = b;
```

SE1 – Level 2

14

Diskussion: Geschachtelte if-Anweisung

vorher: `sum == 0, count == 1, result == 2`

Beispiel 1:

```
if (sum == 0)
    if (count == 0)
        result = 1;
else
    result = 0;
```



- Vorsicht bei geschachtelten `if`-Anweisungen: ein `else`-Zweig bezieht sich immer auf die letzte `if`-Anweisung ohne `else`-Zweig.
- Explizite Klammerung hilft, Fehler zu vermeiden (Beispiel 3)!

Siehe dazu auch Punkt 4.7 der Quelltextkonventionen!

Beispiel 2:

```
if (sum == 0)
    if (count == 0)
        result = 1;
    else
        result = 0;
```

- Beispiel 1 erzeugt durch sein Layout einen falschen Eindruck; Beispiel 2 ist korrekt eingerückt.
- Das hier dargestellte Problem heißt „dangling else“ („else“ ohne Bezug).

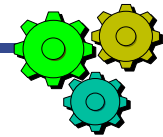
Beispiel 3:

```
if (sum == 0)
{
    if (count == 0)
    {
        result = 1;
    }
}
else
{
    result = 0;
}
```

SE1 – Level 2

15

SE1 Quelltextkonvention zu bedingten Anweisungen



- Bei gewöhnlichen Auswahlanweisungen – `if`-Anweisungen – sollte ein ggf. vorhandenes `else` stets **in einer eigenen Zeile** stehen.
- Allg. gilt: Verwende **immer Blockklammern** bei `if`-Anweisungen, diese sind zwar bei genau einer auszuführenden Anweisung nicht notwendig, erhöhen jedoch die Lesbarkeit des Textes.
- Außerdem ist dies **robuster** gegenüber Änderungen: Wenn beim Eintreten der Bedingung nicht nur eine, sondern auch eine weitere Anweisung ausgeführt werden soll, kann die zweite leicht hinzugefügt werden, ohne dass Klammern eingefügt werden müssen.

Beispiel 3:

```
if (sum == 0)
{
    if (count == 0)
    {
        result = 1;
    }
}
else
{
    result = 0;
}
```

SE1 – Level 2

16

Ein erstes Beispiel für die switch-Anweisung in Java

```
switch (gedruckteTaste)
{
    case 'a':
        bewegeSpielerNachLinks();
        break;

    case 'd':
        bewegeSpielerNachRechts();
        break;

    case 'w':
        bewegeSpielerNachOben();
        break;

    case 's':
        bewegeSpielerNachUnten();
        break;

    case ' ':
        feueRaketeAb();
}
```

case label

- Abhängig von einer auf der Tastatur gedrückten Taste soll in einem Spiel eine von mehreren Prozeduren aufgerufen werden.

- Als case-Label verwenden wir ausschließlich **Konstanten**, häufig vom Typ **int** oder **char**.

Was passiert, wenn

- die **break**-Anweisung fehlt?

SE1 – Level 2

17

Ein weiteres Beispiel für die switch-Anweisung

```
char buchstabe = liesZeichenVonTastatur();
boolean istVokal = false;
switch (buchstabe)
{
    case 'a':
    case 'A':
    case 'e':
    case 'E':
    case 'i':
    case 'I':
    case 'o':
    case 'O':
    case 'u':
    case 'U': istVokal = true;
}
System.out.print(buchstabe + " ist ");
if (!istVokal)
{
    System.out.print('k');
}
System.out.println("ein Vokal.");
```

- Für eine auf der Tastatur gedrückte Taste soll ausgegeben werden, ob sie einen Vokal liefert oder nicht.


Was passiert, wenn

- zwei **case**-Label denselben Wert haben?

SE1 – Level 2

18

Noch ein Beispiel für die switch-Anweisung



```

int monat = liesMonatszahlVomBenutzer();
int tage;
switch (monat)
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        tage = 31;
        break;

    case 4:
    case 6:
    case 9:
    case 11:
        tage = 30;
        break;

    case 2:
        tage = 28;
        break;

    default:
        tage = -1;
}
System.out.println("Der Monat " + monat + " hat " + tage + " Tage.");

```

- Für einen Monat im Jahr, den der Benutzer durch eine ganze Zahl benennen soll, soll ausgegeben werden, wie viele Tage er hat.

Links angedeutet ein Beispiel für den **Kontrollfluss**: Wir geben als Benutzer eine **7** ein.

Was passiert, wenn

- keiner der Fälle zutrifft und die **default**-Anweisung fehlt?

SE1

19

Und noch ein Beispiel für die switch-Anweisung

```

int zahl = liesZehnerpotenzVomBenutzer();
int exponent = 0;
switch (zahl)
{
    case 1000000000: ++exponent;
    case 100000000: ++exponent;
    case 10000000: ++exponent;
    case 1000000: ++exponent;
    case 100000: ++exponent;
    case 10000: ++exponent;
    case 1000: ++exponent;
    case 100: ++exponent;
    case 10: ++exponent;
    case 1: System.out.println(zahl + " = 10^" + exponent); break;
    default: System.out.println(zahl + " ist keine Zehnerpotenz!");
}

```

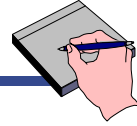
- Der Benutzer soll eine ganze Zahl eingeben, die eine Zehnerpotenz ist. Für eine korrekt eingegebene Zehnerpotenz soll der passende Exponent ausgegeben werden, für alle anderen Zahlen eine Meldung, dass es keine Zehnerpotenz ist.



SE1 - Level 2

20

Selektion mit switch: Auswahlanweisung



- Allgemeines Schema in Java:

```
switch (expression)
{
  case value_1: statements_1;
  case value_2: statements_2;
                break;
  ...
  default: default_statements;
}
```

Seit Java 1.5 kann auch über die Elemente eines Aufzählungstyps „geswitcht“ werden, seit Java 1.7 auch über Strings. Dies ist hier bewusst ausgelassen.

- Die **Auswahlanweisung** (engl.: case statement, switch statement) ist die übliche Form einer **Mehrweg-Verzweigung**. Aufgrund eines Ausdrucks können mehrere **Fälle** unterschieden und behandelt werden.
- Gegenüber der if-Anweisung ist einiges anders:
 - Mehrere Ausdruckstypen** können die Auswahl kontrollieren.
 - Es können **einer oder mehrere Fälle** ausgewählt werden.
 - Statt eines (eindeutigen) else-Falles gibt es einen **Standardfall** für alle nicht explizit benannten Fälle.

Ergänzung der Java Level 1 Syntax um switch

```
Statement:
  Block
  if ( Expression ) Statement [ else Statement ]
  switch ( Expression ) { { SwitchBlockStatementGroup } }
  ...
SwitchBlockStatementGroup:
  { SwitchLabel } { BlockStatement }
SwitchLabel:
  case ConstantExpression :
  default :
```



Zusammengefasst: Auswahlanweisung

- In Java werden alle nach einem passenden Label folgenden Anweisungen durchlaufen; auch wenn der nächste Label oder der **default**-Label erreicht wird.
- Um dies zu vermeiden, kann die Auswahlanweisung mit **break** verlassen werden. Alternativ ist dies auch mit **return** (Verlassen der Methode) oder **throw** (bisher: Abbrechen des Programms) möglich.
- In einer **switch**-Anweisung darf jeder **case**-Label nur einmal vorkommen.
- Wenn kein **case**-Label zutrifft und kein **default**-Label vorhanden ist, wird die gesamte **switch**-Anweisung übersprungen.



Ein (vermutliches) Negativbeispiel:

```
switch (i)
{
    case 1: System.out.println("eins");
    case 2: System.out.println("zwei");
    default: System.out.println("viele");
}
```



Motivation für Schleifen: Einführende Beispiele

- Wir wollen **wiederholt** ein Passwort einlesen, **so lange** die Eingabe noch nicht korrekt ist.

```
String password;
do
{
    System.out.print("Passwort: ");
    password = liesZeileVomBenutzer();
} while (password != _dasPasswort);
```



- Wir möchten **alle** druckbaren Zeichen des ASCII-Zeichensatzes jeweils mit ihrer Ordinalzahl auf der Konsole **ausgeben** lassen.

```
for (char c = 32; c < 127; ++c)
{
    int i = c;
    System.out.println(i + ". ASCII-Zeichen: " + c);
}
```

„Play it again, Sam“: Wiederholung durch Schleifen

- Computer sind besonders gut darin, klaglos die einfachsten Dinge beliebig oft zu wiederholen. Insbesondere sind sie dabei auch sehr schnell.

„Der normale Mensch hat heute mehr Rechenpower zu Hause als ganz Houston, als es ein Apollo-Problem hatte.“

(Gunter Dueck, 2008)

- **Wiederholungsanweisungen** (engl.: iterative statements) ermöglichen, dass Anweisungen keinmal, einmal oder mehrfach ausgeführt werden.
- In imperativen Sprachen werden sie umgangssprachlich auch als **Schleifen** (engl.: loops) bezeichnet.
- Es stellen sich einige Fragen:
 - Wie ist eine Wiederholungsanweisung aufgebaut?
 - Wie wird die Wiederholung kontrolliert?
 - Wo steht der Kontrollmechanismus innerhalb der Schleife?

Die Grundidee: 1x hinschreiben, mehrfach ausführen lassen

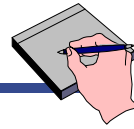
- Wenn wir eine bestimmte Anweisung mehrfach ausführen lassen wollen, können wir dies erreichen, indem wir die Anweisung mehrfach in den Quelltext schreiben:

```
Anweisung A;  
Anweisung A;  
Anweisung A;  
Anweisung A;
```

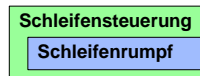
- Dies ist meist nicht zweckmäßig, insbesondere, wenn die Anzahl der Ausführungen nicht bereits beim Schreiben des Quelltextes feststeht.
- Stattdessen schreiben wir die Anweisung nur einmal textuell in den Quelltext und sorgen mit einer umgebenden **Kontrollstruktur** dafür, dass diese Anweisung mehrfach ausgeführt wird.

```
Wiederhole 4 x:  
    Anweisung A;  
Ende der Wiederholung
```

Die Struktur einer imperativen Schleife



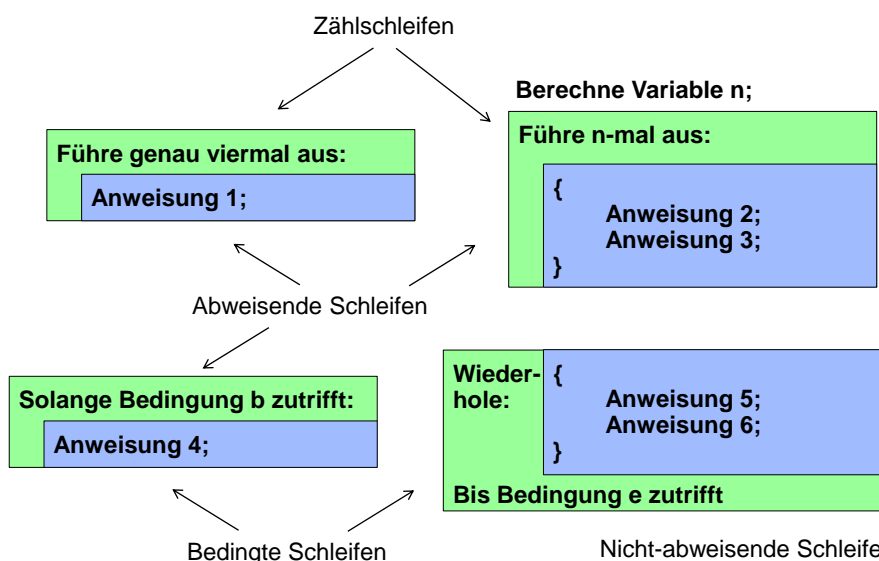
- Wir unterscheiden bei einer Schleife den Schleifenrumpf (engl.: loop body) von der Schleifensteuerung (engl.: loop control):
 - Der **Schleifenrumpf** enthält die zu wiederholenden Anweisungen; üblicherweise ist der Schleifenrumpf ein **Block** (also eine geklammerte Anweisungsfolge, die über eigene Variablen verfügen kann).
 - Die **Schleifensteuerung** steuert die Anzahl der Wiederholungen. Die Schleifensteuerung kann
 - eine feste Anzahl von Wiederholungen definieren
 - oder diese Anzahl von Variablen abhängig machen
 - oder von einer Bedingung, der **Schleifenbedingung**.
- Die Schleifensteuerung können wir uns als eine Art Rahmen oder Klammer um den Schleifenrumpf vorstellen.
- Wichtig: Im Schleifenrumpf können Anweisungen stehen, die Einfluss auf die Schleifensteuerung nehmen.



SE1 – Level 2

27

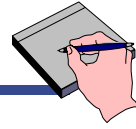
Beispiele für Schleifenarten



SE1 – Level 2

28

Abweisende und nicht-abweisende Schleifen



- Eine Schleife nennen wir **abweisend**, wenn es aufgrund der Schleifensteuerung auch dazu kommen **kann**, dass der Schleifenrumpf gar nicht ausgeführt wird.
- Beispielsweise sind alle Schleifen, bei denen zuerst eine Schleifenbedingung geprüft wird, abweisende Schleifen; denn je nach Ergebnis der ersten Auswertung der Bedingung kann der Schleifenrumpf mindestens einmal ausgeführt werden oder gar nicht.
- Abweisende Schleifen werden teilweise auch als **kopfgesteuerte Schleifen** bezeichnet.
- Wird hingegen der Schleifenrumpf auf jeden Fall mindestens einmal ausgeführt, sprechen wir von einer **nicht-abweisenden** Schleife. Sie wird auch als **fuß- oder endgesteuerte Schleife** bezeichnet.

Bedingte Schleifen



- Die Ausführung einer **bedingten Schleife** ist mit einer **logischen Bedingung** verknüpft.
- Diese Bedingung wird entweder vor (abweisende Schleife) oder nach (endgesteuerte Schleife) **jeder Ausführung** des Schleifenrumpfes **erneut überprüft**.
- Die Bedingung **muss bei jedem** Schleifendurchlauf erneut geprüft werden, weil üblicherweise bei einem Durchlauf Anweisungen ausgeführt werden, die das Ergebnis der Prüfung beeinflussen.
- Unabhängig von der Frage, ob es sich um eine abweisende Schleife handelt oder nicht, kann die Bedingung für eine weitere Ausführung des Schleifenrumpfes **positiv** formuliert sein („Rumpf ausführen, **solange** die Bedingung zutrifft“) oder aus Sicht des Schleifenrumpfes **negativ** („ausführen, **bis** die Bedingung zutrifft“; also **nicht** mehr ausführen, wenn die Bedingung zutrifft).
- Die „Solange-Schleifen“ nennen wir **positiv bedingte Schleifen**, die „Bis-Schleifen“ **zielorientiert bedingte Schleifen**.

Aufpassen: Bedingte Schleifen an einem Beispiel

- **Beispiel:** Ein einzelnes Zeichen soll **so lange** eingelesen werden, **bis** es entweder ein j oder ein n ist (für Ja bzw. Nein).
- Diese fachliche Anforderung ist direkt umsetzbar in Pseudo-Code:
wiederhole
Schleifenrumpf: Einlesen eines Zeichens
bis (ch gleich 'j') oder (ch gleich 'n')
- Das „Problem“ in Java: Es gibt nur positiv bedingte Schleifen; alle bedingten Schleifen in Java werden ausgeführt, **solange** die Schleifenbedingung zutrifft.
- Folglich müssen wir die Bedingung für eine Java-Schleife **negieren**. Aus
wiederhole ... bis (ch == 'j') || (ch == 'n')
 wird dann:
wiederhole, solange (ch != 'j') && (ch != 'n') ...
- Bei dieser **Negation** (logischen Umkehrung) der Bedingung kommen hier die **De Morganschen Regeln** der **Booleschen Algebra** zum Einsatz.

Wdh.: Boolesche Operationen: Einige Rechenregeln

Seien **P**, **Q** und **R** logische Variable, dann gelten die folgenden Identitäten:

Kommutativgesetz:

$P \text{ or } Q \equiv Q \text{ or } P$

$P \text{ and } Q \equiv Q \text{ and } P$

Distributivgesetz:

$(P \text{ and } Q) \text{ or } R \equiv (P \text{ or } R) \text{ and } (Q \text{ or } R)$

$(P \text{ or } Q) \text{ and } R \equiv (P \text{ and } R) \text{ or } (Q \text{ and } R)$

Assoziativgesetz:

$(P \text{ or } Q) \text{ or } R \equiv P \text{ or } (Q \text{ or } R)$

$(P \text{ and } Q) \text{ and } R \equiv P \text{ and } (Q \text{ and } R)$

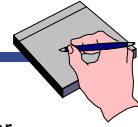
De Morgans Gesetze:

$\text{not } (P \text{ or } Q) \equiv \text{not } P \text{ and } \text{not } Q$

$\text{not } (P \text{ and } Q) \equiv \text{not } P \text{ or } \text{not } Q$

Grundannahme: Der Operator **not** bindet stärker als die Operatoren **and** und **or**.

Zählschleifen



- Bei einer **Zählschleife** ist die Anzahl der Wiederholungen zu Beginn der Schleife festgelegt, entweder durch eine Konstante oder durch die Belegung einer Variablen. Sie sind somit meist abweisend.
- Zählschleifen verfügen üblicherweise über einen **Schleifenzähler** (engl.: loop counter): eine Variable, die im einfachsten Fall die Schleifendurchläufe mitzählt.
- Der Schleifenzähler kann ausschließlich zur Schleifensteuerung dienen, er kann aber auch im Schleifenrumpf verwendet werden.
- Ein Beispiel in Pascal:

```
var i : Integer;
for i := 1 to 10 do
begin
  Writeln('Hallo!');
  Writeln('Durchlauf ', i);
end;
```

Realisierung von Schleifen in Java

Java bietet vier Schleifenkonstrukte zur Realisierung von Wiederholungen, von denen wir vorläufig nur drei betrachten:

While-Schleife: positiv bedingt, abweisend

```
while ( boolean_expression )
  statement
```



Do-While-Schleife: positiv bedingt, endgesteuert

```
do
  statement
while ( boolean_expression )
```

For-Schleife: positiv bedingt, abweisend, ermöglicht u.a. Zählschleifen

```
for ( [ Init_Expr ]; [ Bool_Expr ]; [ Update_Expr ] )
  statement
```

Die for-Schleife in Java

```
for ( [Init_Expr]; [Bool_Expr]; [Update_Expr] )
    statement
```

- Die for-Schleife in Java ist sehr flexibel:
 - Die gesamte Schleifensteuerung kann zwischen den runden Klammern stehen (einschließlich der Deklaration einer Variablen als Schleifenzähler).
 - Init_Expr**: Der Teil der Schleifensteuerung vor dem ersten Semikolon wird einmalig zu Beginn der Schleife ausgeführt.
 - Bool_Expr**: Dann wird die Bedingung zwischen den beiden Semikola geprüft. Wenn diese zutrifft, wird der Schleifenrumpf ausgeführt.
 - Update_Expr**: Nach Ausführung des Schleifenrumpfes wird der Teil nach dem zweiten Semikolon ausgeführt.
 - Anschließend wird erneut die Bedingung geprüft, der Rumpf evtl. ausgeführt und das Update ausgeführt usw.
 - Alle Teile sind optional.



```
for (int i = 0; i < 10; ++i)
{
    System.out.println("Hallo!");
    System.out.println("Durchlauf " + i);
}
```

SE1 – Level 2

35

Endlosschleifen

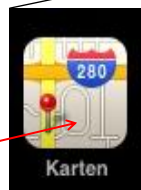
- Endlosschleifen** (engl.: infinite loop) sind meist ungewollt und deshalb unbeliebt.
- Üblicherweise ist die Schleifenbedingung bei einer Endlosschleife falsch gewählt.
- Die einfachsten Endlosschleifen in Java:

```
while (true)
{
    // endlos wiederholt
}
```

oder:

```
for ( ; ; )
```

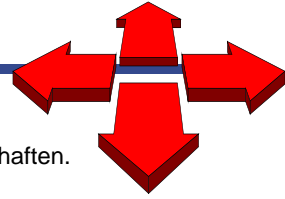
Die Adresse von
Apples
Hauptquartier in
Cupertino, CA:
Infinite Loop 1



SE1 – Level 2

36

Statische und dynamische Eigenschaften



- Programme haben **statische** und **dynamische** Eigenschaften.
- Die statischen Eigenschaften können bei der Übersetzung überprüft werden; dazu zählt auch die **Sichtbarkeit** von Programmelementen.
- Die dynamischen Eigenschaften zeigen sich bei der Ausführung eines Programms; dazu zählt auch die **Lebensdauer** von Variablen und Objekten.
- Ein **Compiler** überprüft u.a. die statischen Eigenschaften von Programmen; wir untersuchen dies näher.

Laufzeit und Übersetzungszeit



- Zwei zentrale Begriffe für Programmiersprachen, die durch Compiler übersetzt werden, sind **Laufzeit** und **Übersetzungszeit**.
- **Übersetzungszeit** (engl.: compile time)
ist die Zeit, in der ein Compiler den in einer Programmiersprache geschriebenen Quelltext in eine ausführbare Form übersetzt. Hier sind die **statischen Eigenschaften** von Programmen relevant:
 - Welche **Syntaxregeln** gibt es? Welche **Sichtbarkeitsregeln** gelten?
 - Wie ist der Quelltext **strukturiert**? Welche Klassen, Methoden etc. gibt es?
 - Wie **lesbar** ist der Quelltext (**Konventionen** etc.)?



Laufzeit und Übersetzungszeit (II)



- **Laufzeit** (engl.: run time)

ist die Zeit, in der ein Computerprogramm im Rechner von seinem Start bis zur Termination (Beendigung) ausgeführt wird. Hier sind die **dynamischen Eigenschaften** von Programmen relevant:

- **Semantik: Was** macht das Programm?
- Wie viele **Objekte** werden **erzeugt**? Welche **Lebensdauer** haben sie?
- Welche **Methoden** werden **aufgerufen**? Welche **Daten manipuliert**?



Veranschaulichungen des Unterschieds

- Am Beispiel von lokalen Variablen können wir den Unterschied zwischen Übersetzungs- und Laufzeit veranschaulichen.
- Was wissen wir zur **Übersetzungszeit** über eine lokale Variable? Wir kennen
 - ihren Namen
 - ihren Typ
 - ihre Sichtbarkeit (nur innerhalb ihrer Methode)
- Was wissen wir meist erst zur **Laufzeit** über eine lokale Variable?
 - ihre Belegungen (können bei jeder Ausführung anders sein)
 - ihre Lebensdauer (wie lange wird sie benötigt?)
 - ihre Adresse (wo steht sie im Speicher?)



Ein weiteres Beispiel für den Unterschied

- „Die return-Anweisung ist immer die letzte Anweisung in einer Methode.“
- Was ist dann mit folgender Methode:

```
public int max(int a, int b)
{
    if (a > b)
    {
        return a;
    }
    else
    {
        return b;
    }
}
```

- Gemeint ist: **Zur Laufzeit** ist die return-Anweisung immer die letzte Anweisung in einer Methode!

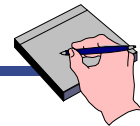
- Zur besseren **Lesbarkeit** könnte eine lokale Variable für das Ergebnis deklariert werden, das nur einmal am Ende der Methode zurückgegeben wird:
- Diese return-Anweisung ist dann auch **statisch** (zur **Übersetzungszeit**) die letzte Anweisung.

```
public int max(int a, int b)
{
    int max;
    if (a > b)
    {
        max = a;
    }
    else
    {
        max = b;
    }
    return max;
}
```

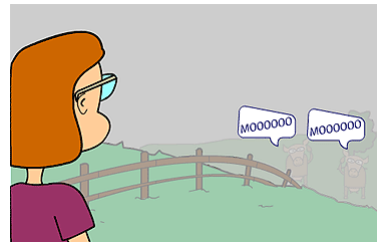
SE1 – Level 2

41

Sichtbarkeitsbereich



- Ein zentraler Begriff der (imperativen) Programmierung ist der **Sichtbarkeitsbereich** (engl.: scope):
 - Jedem Bezeichner in einem Programm wird ein **Bereich** zugeordnet, in dem er angesprochen und benutzt werden kann.
 - Auf den Wert einer sichtbaren Variablen kann z.B. über ihren Namen zugegriffen werden.
- In imperativen und objektorientierten Sprachen ist der Sichtbarkeitsbereich am **Programmtext (statisch)** feststellbar. Der Sichtbarkeitsbereich eines Bezeichners ist gleich der Programmeinheit, in der der Bezeichner deklariert ist.



SE1 – Level 2

42

Sichtbarkeitsbereich objektorientiert

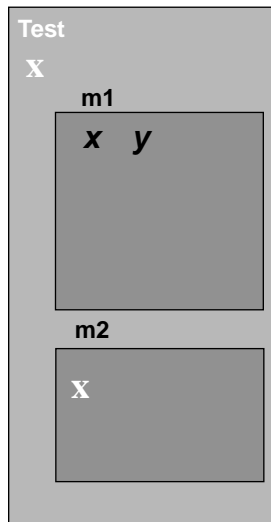
- **Methoden** bilden gegenüber ihrer Umgebung einen eigenen **Sichtbarkeitsbereich**, d.h. sie können lokale Variablen benennen und verwalten.
- Alle im Rumpf einer Methode deklarierten **Variablen** sind **nur im Rest des Methodenrumpfes**, aber nicht außerhalb der Methode sichtbar.
- Die **Umgebung** einer Methode ist in objektorientierten Sprachen ihre **Klasse**, sie bildet den unmittelbar übergeordneten Sichtbarkeitsbereich.
- Die **Exemplarvariablen** einer Klasse sind in allen Methoden der Klasse sichtbar, ebenso wie alle Methoden.
- Die **Sichtbarkeitsbereiche von Klasse und Methode** sind in einander **geschachtelt**.
- In Java können Methoden im Inneren noch weiter durch sog. **Blöcke** in Sichtbarkeitsbereiche unterteilt werden.

Beispiel: Sichtbarkeitsbereiche

```
class Test {
    private int x = 0;

    public void start() {
        m1(); m2();
    }

    private void m1() {
        double x,y;
        ...
        x = 1.5;
        ...
    }
    private void m2() {
        ...
        x = 5;
        ...
    }
}
```



- In der Klasse **Test** sichtbar:

x

- Nur in m1 sichtbar:

x, y

- In m1 verdeckt:

x

- In m2 ist sichtbar:

x

Verdecken von Bezeichnern

- Eine lokale Variable kann den gleichen Bezeichner haben wie eine Variable mit größerer Sichtbarkeit, z.B. eine Exemplarvariable.
- Man sagt dann, dass die lokale Variable die Exemplarvariable „verdeckt“; diese ist dann lokal nicht mehr sichtbar.
- Vorsicht mit dieser Technik des „Verdeckens“, die selten sinnvoll ist und oft zu Fehlern führt.



Hier helfen uns die Quelltextkonventionen: Wenn wir Exemplarvariablen mit führendem Unterstrich benennen (und Parameter und lokale Variablen nicht), kann es nicht zu Überdeckungen kommen.

Klassischer Fehler: Versehentliches Überdecken

```
class Uhrenanzeige
{
    private Nummernanzeige _stunden;
    private Nummernanzeige _minuten;

    public Uhrenanzeige()
    {
        Nummernanzeige _stunden = new Nummernanzeige(24);
        Nummernanzeige _minuten = new Nummernanzeige(60);
    }
}
```



richtig:



```
class Uhrenanzeige
{
    private Nummernanzeige _stunden;
    private Nummernanzeige _minuten;

    public Uhrenanzeige()
    {
        _stunden = new Nummernanzeige(24);
        _minuten = new Nummernanzeige(60);
    }
}
```

Sichtbarkeit der Elemente einer Klasse in Java

In Java kann die Sichtbarkeit von Sprachelementen (hier: Methoden und Exemplarvariablen) durch Modifikatoren (engl.: modifiers) festgelegt werden. Wir kennen bisher folgende Modifikatoren für die Elemente einer Klasse:

public

legt für ein Element der Klasse fest, dass es für Klienten sichtbar und damit öffentlich zugänglich ist. Wir nutzen dies für Methoden, die die Schnittstelle der Klasse bilden sollen.

private

legt für ein Element der Klasse fest, dass es nur innerhalb der Klasse zugänglich ist. Wir nutzen dies meist für Exemplarvariablen und Hilfsmethoden.



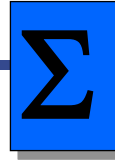
Dazu kommen **protected** und **<default>**, die erst in SE2 thematisiert werden.

Lebensdauer



- Die **Lebensdauer** (engl.: lifetime) einer Variablen oder eines Objektes ist eine *dynamische Eigenschaft*. Lebensdauer bezeichnet die Zeit, in der eine Variable (oder ein ggf. damit verbundenes Objekt) während der Laufzeit **existiert**. Während der Lebensdauer ist einer Variablen (oder einem Objekt) **Speicherplatz** zugewiesen.
- Sichtbarkeit und Lebensdauer können unabhängig voneinander sein, wie in folgender Situation:
 - Eine Exemplarvariable **x** ist statisch deklariert, ein entsprechendes Feld eines Objektes hält zur Laufzeit einen Wert.
 - In einer Methode ist eine gleichnamige lokale Variable **x** deklariert, die die Exemplarvariable verdeckt. Obwohl sie weiter im Speicher existiert, ist die Exemplarvariable während der Ausführung der Methode **nicht über den Namen x sichtbar**.

Zusammenfassung



- Die **strukturierte Programmierung** beschränkt sich auf die **Kontrollstrukturen** Sequenz, Verzweigung und Wiederholung.
- **Schleifenkonstrukte** in imperativen Sprachen sind die einfachste Form für Wiederholungen.
- Wir haben die **Sichtbarkeit** und die **Lebensdauer** von Programmelementen kennen gelernt.
- Die Sichtbarkeit von Programmelementen ist eine **statische** Eigenschaft innerhalb des Programmtextes, die zur **Übersetzungszeit** geprüft werden kann.
- Die Lebensdauer von Programmelementen ist eine **dynamische** Eigenschaft und legt fest, wie lange sie während der **Laufzeit** eines Programms existieren.
- Sichtbarkeit und Lebensdauer hängen teilweise eng zusammen.