Software-Entwicklung 1

V07: Vertiefung Kontrollstrukturen

Prof. Maalej & Team **@maalejw**

Status der 6. Übungswoche

Zeit	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
Vormittag	Gruppe 1 **Erfüllt: 31%**	Gruppe 3 **Erfüllt: 50%**	Gruppe 5 **Erfüllt: %**	Gruppe 6 **Erfüllt: 86%**	Gruppe 8 **Erfüllt: 59%**
Nachmittag	Gruppe 2 **Erfüllt: 50%**	Gruppe 4 **Erfüllt: %**	Vorlesung	Gruppe 7 **Erfüllt: 41%**	

Überblick

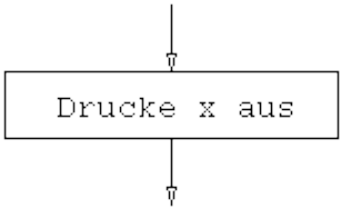
1. **Vertiefung von Kontrollstrukturen**
2. Schleifenmechanismen
3. Sichtbarkeit und Lebenszeit von Variablen

Wiederholung: Kontrollstrukturen

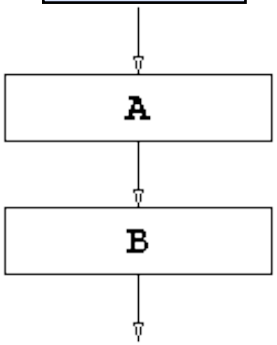
- Kontrollstrukturen der imperativen Programmierung
  - Sequenz
  - Auswahl
  - Wiederholung

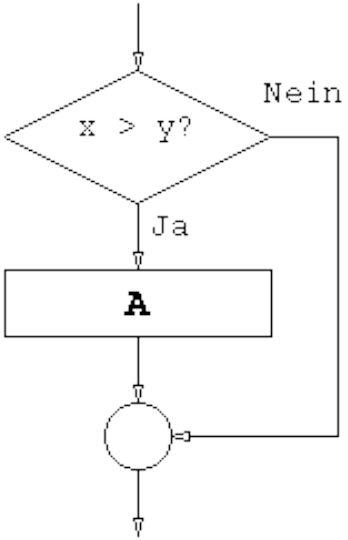
Flussdiagramme zur Darstellung

Einfache Aktion / Anweisung



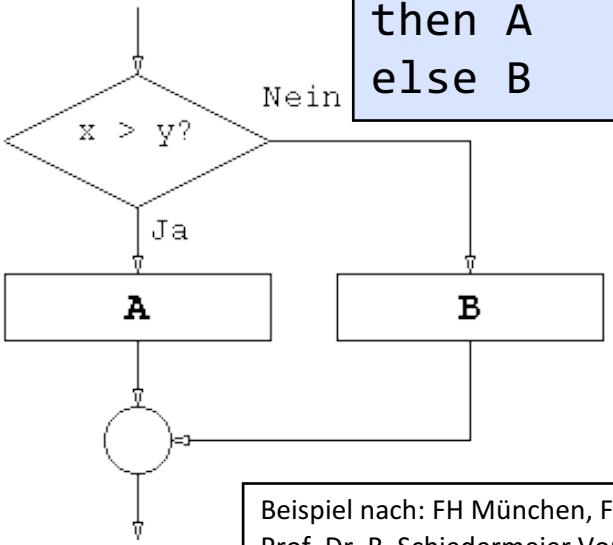
Sequenz
A; B



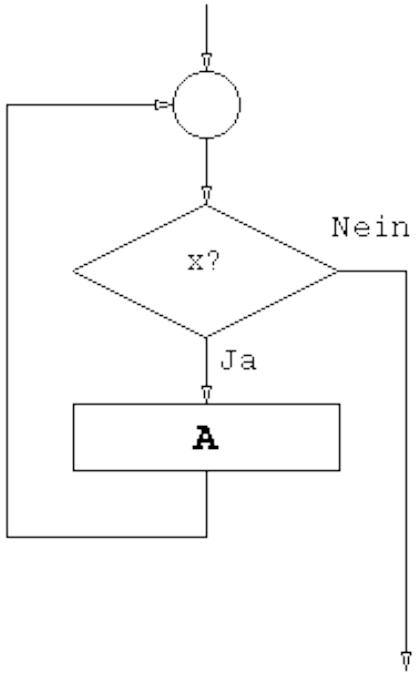


Fallunterscheidung
if x > y
then A

Fallunterscheidung
if x > y
then A
else B



Wiederholung
while x
do A



Beispiel nach: FH München, FB 07 Informatik/Mathematik
Prof. Dr. R. Schiedermeier Vorlesung "Programmieren" im WS 1995/96

goto Statement

- Unbedingte Verzweigung
- Goto führt zu unlesbaren und unzuverlässigen Programmen
- Jede beliebige Reihenfolge von Anweisungen unabhängig von ihrer textlichen Reihenfolge


```
    goto label;
    ...
    ...
    ...
    label:
    ...
    ...
```

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

Blöcke

- Blöcke sind **zusammengesetzte Anweisungen** mit **lokalen Variablen**
- Syntaktisch geklammert; in Java mit geschweiften Klammern { }
- Programmiersprachen mit Blöcken heißen auch blockstrukturiert
- **Ausblick**: Blöcke bilden einen eigenen Sichtbarkeitsbereich

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

Geschachtelte if-Anweisung

- **Vorsicht:** ein `else`-Zweig bezieht sich immer auf die letzte `if`-Anweisung ohne `else`-Zweig
- Dieses Problem heißt „dangling else“ („else“ ohne Bezug)

Beispiel 1:

```
result = 0;
if (false)
    if (true)
        result = 1;
else
    result = 2;
```

Beispiel 1: Erzeugt durch sein Layout einen falschen Eindruck;

Beispiel 2:

```
result = 0;
if (false)
    if (true)
        result = 1;
   else
        result = 2;
```

Beispiel 2: Korrekt eingerückt

Lösung: Geschachtelte if-Anweisung

- Explizite Block-Klammerung hilft, Fehler zu vermeiden
- `if` und `else` Anweisungen sollten in einer eigenen Zeile stehen

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

Selektion mit switch-Anweisung

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

- **Auswahlanweisung** (engl.: case statement, switch statement) ist eine **Mehrweg-Verzweigung**
- Mehrere **Fälle** können unterschieden behandelt werden
- Unterschiede zur if-Anweisung:
  - **Mehrere Ausdruckstypen** können die Auswahl kontrollieren
  - Es können **einer oder mehrere Fälle** ausgewählt werden
  - Statt else-Falles gibt es einen **Standardfall** für alle nicht benannten Fälle

(I) Beispiel für switch-Anweisung


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

case 'd':
    bewegeSpielerNachRechts();
    break;

case 'w':
    bewegeSpielerNachOben();
    break;

case 's':
    bewegeSpielerNachUnten();
    break;

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

case label

Abhängig von der Tastatureingabe soll ein Spieler bewegt werden

`case`-Labels sind ausschließlich Konstanten; häufig vom Typ int oder char

Was passiert, wenn die break-Anweisung fehlt?

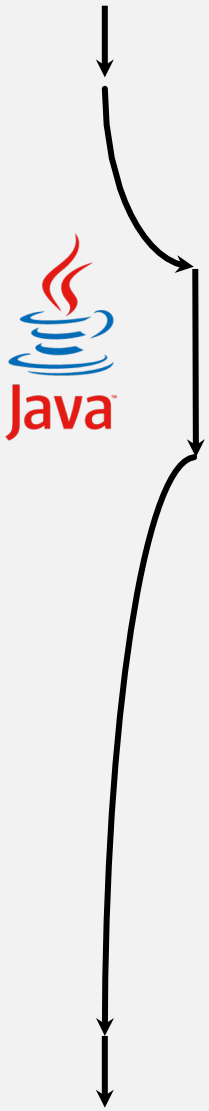
(II) Beispiel für switch-Anweisung


```
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 Tastatureingabe soll ausgegeben werden, ob sie einen Vokal liefert oder nicht

Was passiert, wenn zwei `case`-Label denselben Wert haben?

(III) Beispiel für 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?

(IV) Beispiel für 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 Zehnerpotenz als Zahl eingeben.

Für eine korrekt eingegebene Zehnerpotenz soll der passende Exponent ausgegeben werden,

für alle anderen Zahlen eine Meldung, dass es keine Zehnerpotenz ist.

Negativbeispiel für switch-Anweisung

- Abhängig von dem Wert von i wird eine andere Printanweisung ausgeführt


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

Zusammenfassung: switch-Anweisung

1. Alle nach einem passenden Label folgenden werden durchlaufen. Auch über die nächsten Label hinaus.

2. Eine Auswahlanweisung kann mit **break** verlassen werden. Alternativ ist dies auch mit **return** möglich.

3. In einer **switch**-Anweisung darf jeder **case**-Label nur einmal vorkommen.

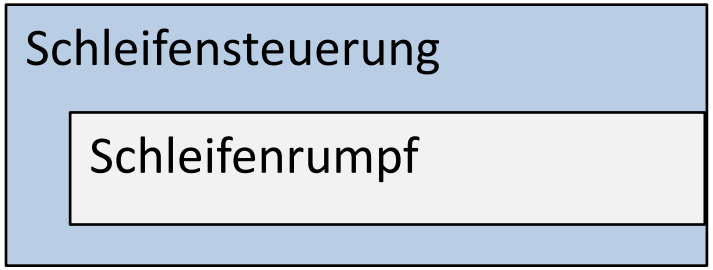
4. Wenn kein **case**-Label zutrifft und kein **default**-Label vorhanden ist, wird die gesamte **switch**-Anweisung übersprungen.

Überblick

1. Vertiefung von Kontrollstrukturen
2. **Schleifenmechanismen**
3. Sichtbarkeit und Lebenszeit von Variablen

Struktur von imperativen Schleifen

- **Schleifensteuerung**
  - Anzahl der Wiederholungen
    - Feste Anzahl
    - Abhängig von Variablen
    - Abhängig von Schleifenbedingung



- **Schleifenrumpf**
  - Enthält die zu wiederholenden Anweisungen
  - Üblicherweise ist der Schleifenrumpf ein Block

- Schleifensteuerung ist wie ein Rahmen oder Klammer um den Schleifenrumpf
- **Wichtig:** Der Schleifenrumpf kann Einfluss auf die Schleifensteuerung nehmen

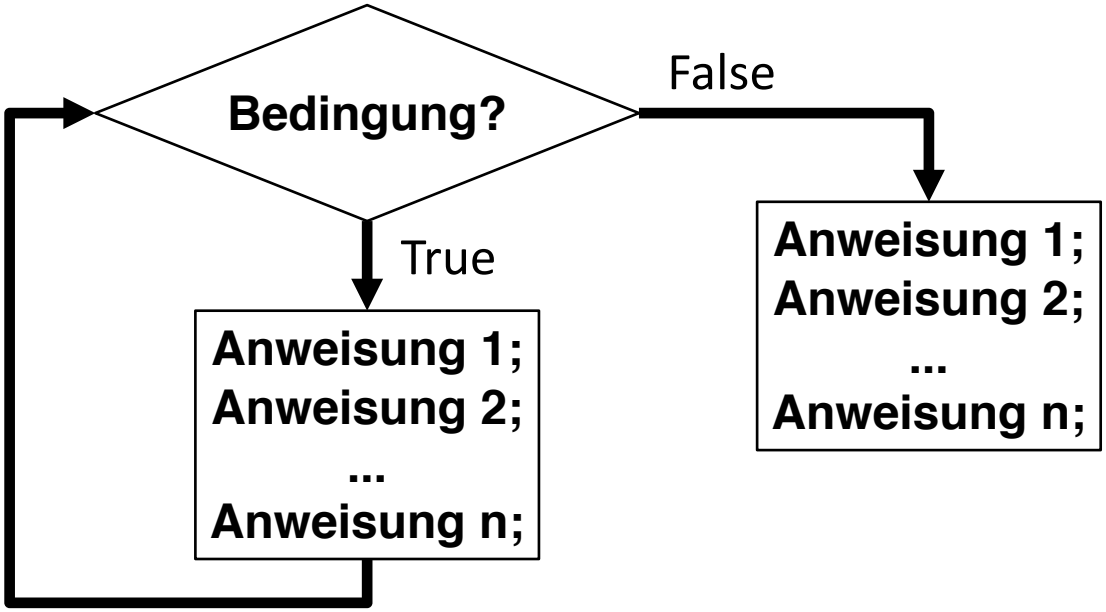
Abweisende und Annehmende Schleifen

- **Abweisend**, wenn der Schleifenrumpf nicht zwangsläufig ausgeführt wird; Es wird zuerst eine Schleifenbedingung geprüft
- Auch **kopfgesteuerte** Schleife genannt
- Beispiel: while oder for-Schleife

- **Annehmend,** wenn der Schleifenrumpf bedingungslos mindestens einmal ausgeführt wird
- Auch **fuß- oder endgesteuerte** Schleife genannt
- Beispiel Do-While-Schleife

Bedingte Schleifen

- Bedingt, wenn die Ausführung des Rumpfs mit einer **logischen Bedingung** verknüpft ist
- Bedingung wird entweder **vor** (abweisende Schleife) oder **nach** (annehmende Schleife) jeder Ausführung des Schleifenrumpfes erneut überprüft
- Bedingung wird bei **jedem Schleifendurchlauf** erneut **geprüft**, weil bei der Ausführung Einfluss auf das Ergebnis der Prüfung genommen wird



Aufpassen: Bedingte Schleifen an einem Beispiel

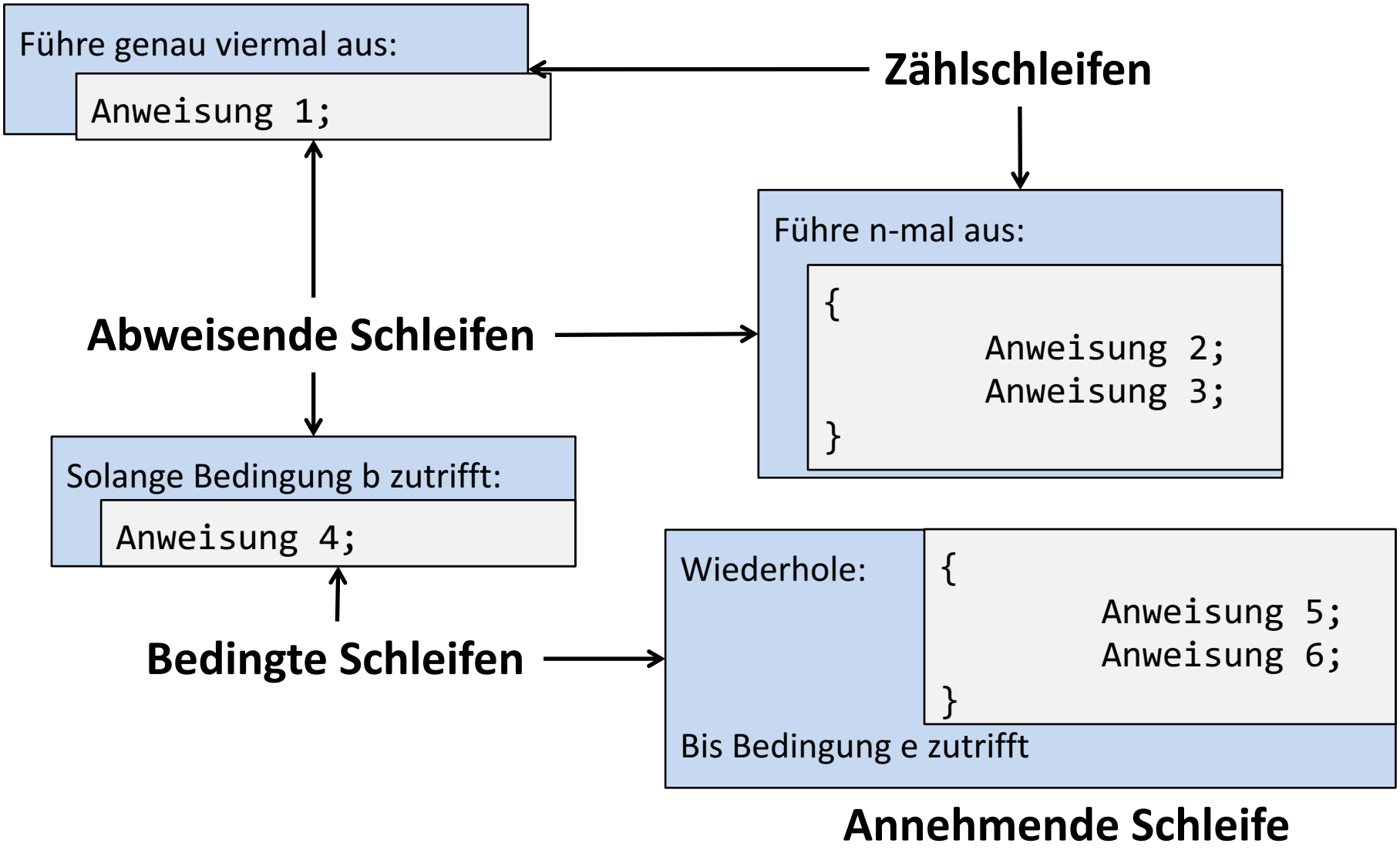
- Beispiel: Ein einzelnes Zeichen soll so lange eingelesen werden, bis es entweder ein j oder ein n ist (für Ja bzw. Nein).
- Diese fachliche Anforderung ist direkt umsetzbar in Pseudo-Code:
  - wiederhole
    - Schleifenrumpf: Einlesen eines Zeichens ch
  - bis (ch gleich 'j') oder (ch gleich 'n')
- 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.

Zählschleifen

- Zählschleife, wenn die Anzahl der Wiederholungen zu Beginn der Schleife fest steht
- Meist abweisend
- Zählschleifen verfügen üblicherweise über einen Schleifenzähler (engl.: loop counter)
- 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;
```

Beispiele für Schleifenarten



Realisierung von Schleifen

- 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
```

Do-While-Schleife

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

1. **do** leitet den Anweisungsblock ein und wird bedingungslos ausgeführt
2. Nach dem Anweisungsblock kommt der **while**-Teil mit der Abbruchbedingung

- Bei der do-while-Schleife steht die **Abbruchbedingung unten**
- Do-While schleifen sind annehmende Schleifen und werden als **rumpf- oder fußgesteuerte Schleifen** bezeichnet

for-Schleife


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

- Schleifensteuerung steht zwischen runden Klammern (einschließlich der Deklaration einer Variablen als Schleifenzähler)
- **Init_Expr:**
  Wird einmalig zu Beginn der Schleife ausgeführt
- **Bool_Expr:** Die Bedingung, die für ein Ausführen des Rumpfes geprüft wird
- **Update_Expr:** Nach der Ausführung des Schleifenrumpfes wird ein Update ausgeführt
- Es wird erneut die Bedingung geprüft, der Rumpf evtl. ausgeführt und das Update ausgeführt usw.
- Alle Teile sind optional

Schleifenbeispiele

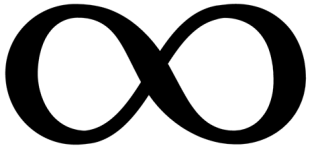
Wiederholt ein Passwort einlesen, so lange die Eingabe noch nicht korrekt ist:

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

Alle Ziffern werden auf der Konsole ausgegeben:

```
for (int i = 0; i < 10; ++i)
{
    System.out.println(i + " ist eine Ziffer.");
}
```

Endlosschleifen



- Meistens ist die Schleifenbedingung bei einer Endlosschleife falsch gewählt

Endlosschleifen in Java:

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

Oder:

```
for ( ; ; );
```

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

Zusammenfassung: Schleifenmechanismen

1. **Schleifenkonstrukte** in imperativen Sprachen sind die einfachste Form für Wiederholungen.

2. Java bietet vier Schleifenkonstrukte zur Realisierung von Wiederholungen, von denen wir vorläufig die: **while, do-while, for** Schleifen betrachtet haben.

3. Die **Schleifensteuerung** regelt die Anzahl der Wiederholungen. Der **Schleifenrumpf** beinhaltet die auszuführenden Anweisungen.

Überblick

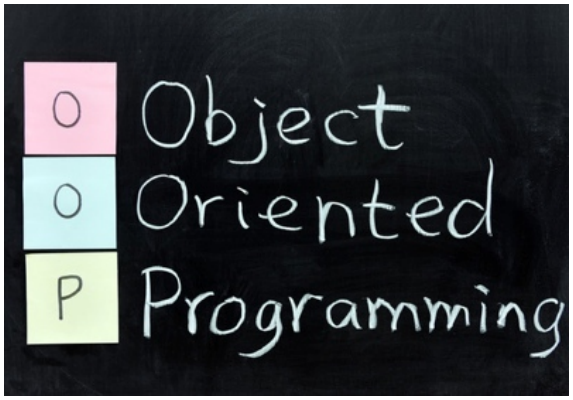
1. Vertiefung von Kontrollstrukturen
2. Schleifenmechanismen
3. **Sichtbarkeit und Lebenszeit von Variablen**

Sichtbarkeitsbereich

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

Sichtbarkeitsbereich Objektorientiert

- **Methoden** bilden einen eigenen **Sichtbarkeitsbereich für lokale Variablen**
- Die **Umgebung** einer Methode ist in objektorientierten Sprachen ihre **Klasse**, sie bildet den 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 **Blöcke** in Sichtbarkeitsbereiche unterteilt werden

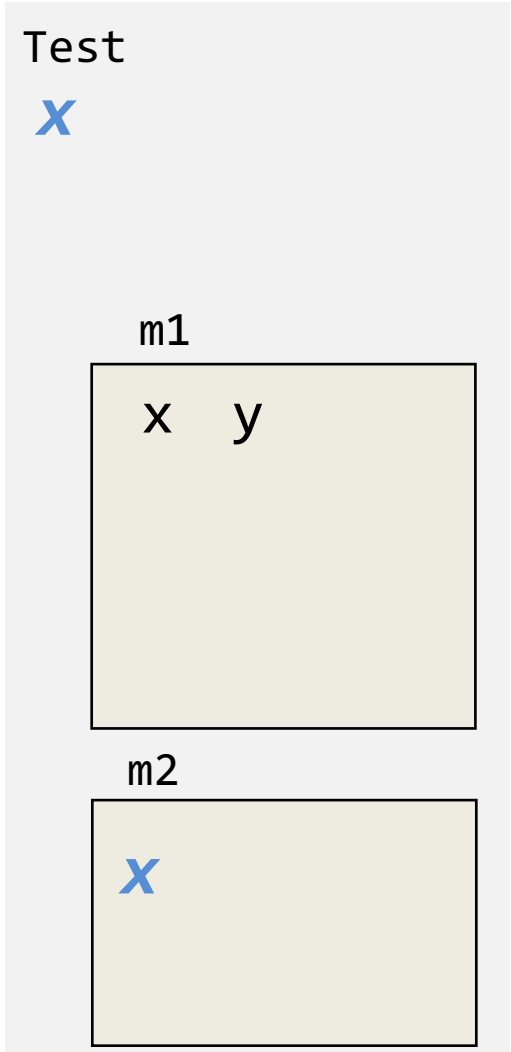


Beispiel: Sichtbarkeitsbereiche

```
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

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)
- Die lokale Variable “verdeckt“ dann die Exemplarvariable; diese ist dann lokal nicht mehr sichtbar

Wenn wir Exemplarvariablen mit führendem Unterstrich benennen (und Parameter und lokale Variablen nicht), kann es nicht zu Überdeckungen kommen.

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

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

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

- Zeit, in der eine Variable (oder ein ggf. damit verbundenes Objekt) während der Laufzeit **existiert**
- Während der Lebensdauer ist einer Variablen **Speicherplatz** zugewiesen
- Sichtbarkeit und Lebensdauer können unabhängig voneinander sein (Beispiel Verdeckung)
- Bei **Objekten** in Java ist die **Lebensdauer** davon abhängig, ob noch Referenzen auf sie existieren

Zusammenfassung: Sichtbarkeit und Lebensdauer

1. Wir haben die **Sichtbarkeit** und die **Lebensdauer** von Programmelementen kennen gelernt.

2. Die Sichtbarkeit von Programmelementen ist eine **statische** Eigenschaft innerhalb des Programmtextes, die zur **Übersetzungszeit** geprüft werden kann.

3. Die Lebensdauer von Programmelementen ist eine **dynamische** Eigenschaft und legt fest, wie lange sie während der **Laufzeit** eines Programms existieren.