Überblick

Funktionen und Prozeduren

- In der funktionalen Programmierung werden Algorithmen als Funktionen dargestellt.
- Das imperative Pendant dazu ist die Prozedur, die sogar ein etwas allgemeineres Konzept darstellt (genaugenommen ist eine Funktion eine Prozedur ohne Seiteneffekte).
- Das Konzept der Prozedur dient (wie das der Funktion auch) zur Abstraktion von Algorithmen.
  - Durch Parametrisierung wird von der Identität der Daten abstrahiert: die Berechnungsvorschriften werden mit abstrakten Parametern formuliert – konkrete Eingabedaten bilden die aktuellen (Parameter-) Werte.
  - Durch Spezifikation des (Ein-/Ausgabe-) Verhaltens wird von den Implementierungsdetails abstrahiert: Vorteile sind
    - **Örtliche Eingrenzung (Locality)**: Die Implementierung einer Abstraktion kann verstanden oder geschrieben werden ohne die Implementierungen anderer Abstraktionen kennen zu müssen.
    - **Änderbarkeit (Modifiability)**: Jede Abstraktion kann reimplementiert werden ohne dass andere Abstraktionen geändert werden müssen.
    - **Wiederverwendbarkeit (Reusability)**: Die Implementierung einer Abstraktion kann beliebig wiederverwendet werden.

Funktionen und Prozeduren

- Beispiel: Algorithmus *Strecke* als Prozedur (in Pseudo-Code).

```
ALGORITHM strecke

  INPUT: m, t, k: REAL
  OUTPUT: erg: REAL

  BEGIN
    VAR b : REAL;
    b = k / m;
    erg = (b * (t * t))/ 2.0;
  END
```


Funktionen und Prozeduren

- Beispiel: Algorithmus zur Berechnung der Funktion $f : (x)$ für $x \neq -1$ mit $f : \mathbb{R} \rightarrow \mathbb{R}$ und $f(x) = (x + 1 + \frac{1}{x+1})^2$ für $x \neq -1$ (siehe Folie 2) als Prozedur (in Pseudo-Code).

```
ALGORITHM f

  INPUT: x: REAL
  OUTPUT: erg: REAL

  BEGIN
    VAR y : REAL
    y = x + 1;
    y = y + 1/y;
    erg = y * y;
  END
```

Prozeduren in Java

- In Java werden Prozeduren durch statische Methoden realisiert.
- Eine Methode wird definiert durch den Methodenkopf:
  `public static` `<typ> <name>(<parameterliste>)`
  und den Methodenrumpf, einen Block, der sich an den Methodenkopf anschließt.
- Als besonderer Ergebnis-Typ einer Methode ist auch **`void`** möglich (vgl. den Typ `unit` in SML). Dieser Ergebnis-Typ bedeutet, die Methode gibt *kein* Ergebnis zurück. Der Sinn einer solchen Methode liegt also ausschließlich in den Nebeneffekten.
- Eine Methode, die Nebeneffekte hat, die den weiteren Programmverlauf beeinflussen, sollte als Ergebnis-Typ stets **`void`** haben. (Manchmal wählt man als Ergebnistyp auch **`boolean`** und zeigt damit an, ob der beabsichtigte Nebeneffekt erfolgreich war.)

Prozeduren in Java

Beispiel: Algorithmus *Strecke*:

```
public class Streckenberechnung
{
  /**
   * Methode zur Berechnung der zurueckgelegten Strecke
   * nach Einwirken der Kraft <code>k</code>
   * auf einen Koerper der Masse <code>m</code>
   * fuer die Zeitdauer <code>t</code>.
   *
   * @param m Masse des Koerpers
   * @param t Zeitdauer
   * @param k Kraft
   * @return zurueckgelegte Strecke
   */
  public static double strecke(double m, double t, double k)
  {
    double b = k / m;
    return 0.5 * b * (t * t);
  }
}
```

Prozeduren in Java

Beispiel: Algorithmus für die Funktion f:

```
public class Funktionsberechnung
{
  /**
   * Methode zur Berechnung der Funktion
   * <i>f(x) = ((x + 1) + 1 / (x + 1))<sup>2</sup></i>.
   *
   * @param x der Eingabewert
   * @return f(x)
   */
  public static double f(double x)
  {
    double y = x + 1;
    y = y + 1/y;
    y = y * y;
    return y;
  }
}
```


Methodenaufruf

Wie wir im Abschnitt über Anweisungen gesehen haben, bildet ein Methodenaufruf eine Anweisung. Ein Methodenaufruf kann also überall da stehen, wo eine Anweisung möglich ist.
Beispiel für einen Methodenaufruf:

```
public class HelloWorld
{
  public static void gruessen(String gruss)
  {
    System.out.println(gruss);
      // auch println(...); ist natuerlich ein Methodenaufruf
  }

  public static void main(String[] args)
  {
    gruessen("Hello, World!");
  }
}
```

Die `main`-Methode

- In Programmbeispielen haben wir bereits die `main`-Methode gesehen. Die `main`-Methode ermöglicht das selbständige Ausführen eines Programmes.
- Der Aufruf `java KlassenName` führt die `main`-Methode der Klasse `KlassenName` aus.
- Die `main`-Methode hat immer einen Parameter, ein `String`-Array. Dies ermöglicht das Verarbeiten von Argumenten, die über die Kommandozeile übergeben werden.

Parameter für die `main`-Methode

Beispiel für einen Zugriff der `main`-Methode auf das Parameterarray:

```
public class Gruss
{
  public static void gruessen(String gruss)
  {
    System.out.println(gruss);
  }

  public static void main(String[] args)
  {
    gruessen(args[0]);
  }
}
```

Dadurch vielfältigere Verwendung möglich:

- `java Gruss "Hello, World!"`
- `java Gruss "Hallo, Welt!"`
- `java Gruss "Servus!"`

Verarbeitung von Parametern

Zur Verarbeitung der Parameter von Prozeduren kennen Programmiersprachen zwei grundsätzliche Möglichkeiten:

- call-by-value
  Im Aufruf `methodenName(parameter)` wird für `parameter` im Methoden-Block ein neuer Zettel angelegt, auf den der Wert von `parameter` geschrieben wird. Auf diese Weise bleibt der ursprüngliche Zettel `parameter` von Anweisungen innerhalb der Methode unberührt.
- call-by-reference
  Im Aufruf `methodenName(parameter)` wird der Zettel `parameter` weiter verwendet. Wenn innerhalb der Methode der Wert von `parameter` verändert wird, hat das auch Auswirkungen außerhalb der Methode.
  call-by-reference ist daher eine potentielle Quelle unbeabsichtigter Seiteneffekte.

In Java: call-by-value

Java wertet Parameter call-by-value aus. Für den Aufruf der Methode `swap` im folgenden Beispiel werden also Kopien der Variablen `x` und `y` angelegt.

```
public class Exchange
{
  public static void swap(int i, int j)
  {
    int c = i;
    i = j;
    j = c;
  }

  public static void main(String[] args)
  {
    int x = 1;
    int y = 2;
    swap(x,y);
    System.out.println(x); // Ausgabe?
    System.out.println(y); // Ausgabe?
  }
}
```

Ausblick: call-by-reference-Effekt bei Objekten

Wenn der Parameter kein primitiver Typ ist, sondern ein Objekt (also z.B. ein Array – was genau Objekte sind, betrachten wir aber später) dann wird zwar in der Methode ebenfalls mit einer Kopie des Parameters gearbeitet, aber es handelt sich um eine Kopie der Speicheradresse, also eine zweite Zugriffsmöglichkeit für den selben Zettel.

```
public static void changeValues(int[] werte, int index, int wert)
{
  werte[index] = wert;
}

public static void main(String[] args)
{
  int[] werte = {0, 1, 2};
  changeValues(werte, 1, 3);
  System.out.println(werte[1]); // Ausgabe: 3
}
```

Obwohl also auch hier die Parameterauswertung nach dem Prinzip call-by-value erfolgt, ist der Effekt der gleiche wie bei call-by-reference. Wir werden auf diesen Effekt zurückkommen.

Überblick

Bedingte Anweisungen

Motivation:

- In vielen Algorithmen benötigt man eine Fallunterscheidung zur Lösung des gegebenen Problems.
- Beispiel: Algorithmus *Wechselgeld* (Falls . . . dann . . .).
- In der (rein) funktionalen Programmierung gibt es daher die Möglichkeit der *fall-basierten* Definition von Funktionen:

```
fun wg(r) = if r = 100 then []
       else if 100 - r >= 5 then 5 :: (wg(r + 5))
       else if 5 > 100 - r andalso 100 - r >= 2
                            then 2 :: (wg(r + 2))
       else (* 2 > 100 - r andalso 100 - r >= 1 *)
            1 :: (wg(r + 1));
```


Bedingte Anweisungen

- Im einfachsten Fall (Pseudo-Code):
  `IF <Bedingung> THEN <Ausdruck1> ELSE <Ausdruck2> ENDIF`
- Dies entspricht einer echten *Fallunterscheidung*:
  - Ist `<Bedingung>` wahr, dann wird `<Ausdruck1>` ausgewertet.
  - Ist `<Bedingung>` falsch, dann wird `<Ausdruck2>` ausgewertet.
  - **Achtung:** `<Ausdruck1>` und `<Ausdruck2>` müssen dabei (in der funktionalen Programmierung, z.B. bei SML) den selben Typ haben!!!

Bedingte Anweisungen

- In der imperativen Programmierung ist dies nicht der Fall:
  - In den verschiedenen Zweigen stehen *Anweisungen* anstelle von Ausdrücken.
  - Damit entfällt die Forderung nach gleichen Typen.
- Man spricht daher von *bedingten Anweisungen*.
- Die Fallunterscheidung ist ein Spezialfall der bedingten Anweisung.
- Die einfachste Form von bedingten Anweisungen ist:

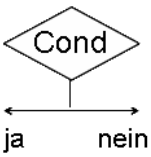
```
IF <Bedingung> THEN <Anweisungsfolge> ENDIF
```

- Bedingte Anweisungen können beliebig oft verzweigt werden:

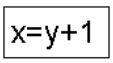
```
IF <Bedingung1> THEN <Anweisungsfolge1>
ELSE IF <Bedingung2> THEN <Anweisungsfolge2>
⋮
ELSE <AnweisungsfolgeN>
ENDIF
```

- Bei mehr als einem Zweig spricht man auch von *bewachten Anweisungen*.

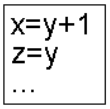
Graphische Darstellung: Kontrollflussdiagramme



Fallunterscheidung bzgl. der Bedingung `Cond`

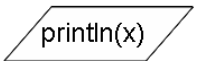


Zuweisung



Block

```
{
  x = y + 1;
  z = y;
}
```

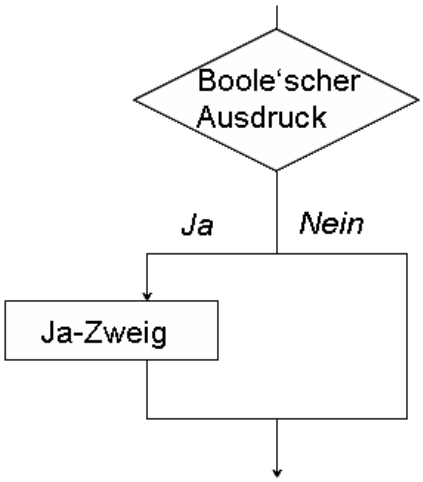


Ein- / Ausgabe

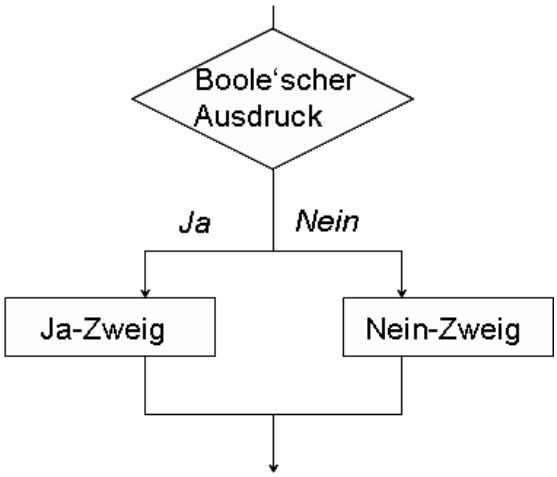
Sequentielle Abfolge

Kontrollflussdiagramme für bedingte Anweisungen

Einfache bedingte Anweisung:



Zweifache bedingte Anweisung:



Bedingte Anweisungen und Rekursion

- Rekursion ist ein nützliches und elegantes Entwurfskonzept für Algorithmen.
- Rekursion entsteht, wenn im Rumpf einer Methode diese selbst wieder aufgerufen wird.
- Damit die Rekursion terminiert benötigt man in der Regel einen oder mehrere Basisfälle (neben einem oder mehreren Rekursionsfällen).
- Diese verschiedenen Fälle werden typischerweise durch bedingte Anweisungen realisiert.

Bedingte Anweisungen und Rekursion

- Beispiel: Algorithmus zur Berechnung der Fakultät für eine natürliche Zahl $n \in \mathbb{N}$ in Pseudo-Code:

```
ALGORITHM fakultaet

  INPUT: n: INTEGER,
  OUTPUT: erg: INTEGER,

  BEGIN
    IF(n==0)
    THEN erg = 1;
    ELSE erg = n * fakultaet(n-1);
  END
```


Bedingte Anweisungen in Java

- Java erlaubt zwei Formen von bedingten Anweisungen (auch: **`if`**-*Schleifen*):
  1. Eine einfache Verzweigung:

     ```
     if (<Bedingung>)
        <Anweisung>
     ```

  2. Eine zweifache Verzweigung:

     ```
     if (<Bedingung>)
        <Anweisung1>
     else
        <Anweisung2>
     ```

  wobei

  - `<Bedingung>` ein Ausdruck vom Typ **`boolean`** ist,
  - `<Anweisung>`, `<Anweisung1>` und `<Anweisung2>` jeweils einzelne Anweisungen, bzw. ein Block von Anweisungen darstellen.

Bedingte Anweisungen in Java

- Beispiel: Der Algorithmus zur Berechnung der Fakultät einer natürlichen Zahl $n \in \mathbb{N}$ kann in Java durch folgende Methode umgesetzt werden:

```
public static int fakultaet(int n)
{
  int erg;
  if(n==0)
  {
    erg = 1;
  }
  else
  {
    erg = n * fakultaet(n-1);
  }
  return erg;
}
```


Bedingte Anweisungen in Java

- Bedingte Anweisung mit mehr als zwei Zweigen müssen in Java durch Schachtelung mehrere **`if`**-Schleifen ausgedrückt werden:

```
if (<Bedingung1>)
   <Anweisung1>
else if (<Bedingung2>)
   <Anweisung2>

:

else if (<BedingungN>)
   <AnweisungN>
else
   <AnweisungN+1>
```

Dangling Else

- Gegeben:

```
if (a)
  if (b)
    s1;
else
  s2;
```

- Frage: Zu welchem **`if`**-Statement gehört der **`else`**-Zweig?

Dangling Else

- Antwort: Zur inneren Verzweigung **`if`** `(b)`. (Die (falsche!) Einrückung ist belanglos für den Compiler und verführt den menschlichen Leser hier, das Programm falsch zu interpretieren.)
- Tipp: Immer Blockklammern verwenden!

```
if (a)
{
   if (b)
   {
      s1;
   }
   else
   {
      s2;
   }
}
```

Mehrfachanweisungen

- Bei Mehrfachanweisungen im **`else`**-Zweig sind Blockklammern sehr wichtig, sonst kann es zu falschen Ergebnissen kommen.
- Beispiel:
  Ein Kunde einer Bank hebt einen Betrag (Variable `betrag`) von seinem Konto (Variable `kontoStand` für den aktuellen Kontostand) ab. Falls der Betrag nicht gedeckt ist, wird eine Überziehungsgebühr fällig. Die fälligen Gebühren werden über einen bestimmten Zeitram akkumuliert (Variable `gebuehren`) und am Ende des Zeitraums in Rechnung gestellt. Was ist falsch in folgender Berechnung?

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


Mehrfachanweisungen

- Problem im vorherigen Beispiel:
  Überziehungsgebühr wird immer verlangt, auch wenn das Konto gedeckt ist.
- Lösung: Blockklammern setzen.

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

- Nun wird die Überziehungsgebühr nur verlangt, wenn das Konto nicht gedeckt ist.

Spezielle Mehrfachverzweigung in Java

- In Java gibt es eine weitere Möglichkeit, spezielle Mehrfachverzweigungen auszudrücken.
- Die sog. **`switch`**-Anweisung funktioniert allerdings etwas anders als bedingte Anweisungen.
- Die **`switch`**-Anweisung ist unter gewissen Umständen ähnlich zum Pattern-Matching.
- Syntax:

```
switch (ausdruck)
{
  case konstante1 : anweisung1
  case konstante2 : anweisung2

    ⋮

  default : anweisungN
}
```


Spezielle Mehrfachverzweigung in Java

- Bedeutung:
  - Abhängig vom Wert des Ausdrucks `ausdruck` wird die *Sprungmarke* angesprungen, deren Konstante mit dem Wert von `ausdruck` übereinstimmt.
  - Die Konstanten und der Ausdruck müssen den selben Typ haben.
  - Die Anweisungen nach der Sprungmarke werden ausgeführt.
  - Die optionale **`default`**-Marke wird dann angesprungen, wenn keine passende Sprungmarke gefunden wird.
  - Fehlt die **`default`**-Marke und wird keine passende Sprungmarke gefunden, so wird keine Anweisung innerhalb der **`switch`**-Anweisung ausgeführt.

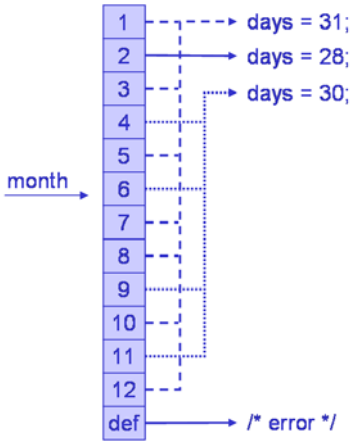
Spezielle Mehrfachverzweigung in Java

- Besonderheiten:
  - Der Ausdruck `ausdruck` darf nur vom Typ **`byte`**, **`short`**, **`int`** oder **`char`** sein.
  - Die **`case`**-Marken sollten alle verschieden sein, müssen aber nicht.
  - **Achtung:** Wird zu einer Marke gesprungen, werden alle Anweisungen hinter dieser Marke ausgeführt. Es erfolgt **keine** Unterbrechung, wenn das nächste Label erreicht wird, sondern es wird dort fortgesetzt! Dies ist eine beliebte Fehlerquelle!
  - Eine Unterbrechung kann durch die Anweisung **`break;`** erzwungen werden. Jedes **`break`** innerhalb einer **`switch`**-Anweisung verzweigt zum Ende der **`switch`**-Anweisung.
  - Nach einer Marken-Definition **`case`** muss nicht zwingend eine Anweisung stehen.

Beispiel

```
switch (month)
{
  case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
      days = 31; break;
  case 4: case 6: case 9: case 11:
      days = 30; break;
  case 2:
      if(leapYear)
      {
        days = 29;
      }
      else
      {
        days = 28;
      }
      break;
  default:
      /* error */
}
```

Sprungtabelle:



Wiederholungsanweisungen (Schleifen)

Motivation:

- Im Algorithmus *Wechselgeld* hatten wir eine Anweisung der Form **Solange** . . . : . . .
- Dabei handelt es sich um eine sog. *bedingte Wiederholungsanweisung (Schleife)*.
- Allgemeine Form:
  `WHILE` `<Bedingung>` **`DO`** `<Anweisung>` **`ENDDO`**
- `<Bedingung>` heißt Schleifenbedingung, `<Anweisung>` heißt Schleifenrumpf und kann natürlich aus mehreren Anweisungen bestehen.

Wiederholungsanweisungen (Schleifen)

Motivation (cont):

- Der Algorithmus zur Berechnung der Fakultät für eine natürliche Zahl $n \in \mathbb{N}$ (Methode `fakultaet`)ist rekursiv definiert.
- Bei Aufruf der Methode `fakultaet` für ein $n \in \mathbb{N}$ wird (in vereinfachter Form) das Ergebnis `erg` letztlich durch folgende Anweisungsfolge berechnet:

```
erg := 1;
erg := 1 * erg;
erg := 2 * erg;
⋮
erg := n * erg;
```

Wiederholungsanweisungen (Schleifen)

Motivation (cont):

- Diese Folge von Zuweisungen könnte man wie folgt imperativ notieren: **Führe für** $i = 1,\ldots, n$ **nacheinander aus** : `erg := i * erg`.
- Dabei handelt es sich um eine sog. *gezählte Wiederholungsanweisung (Schleife)* (auch *Laufanweisung*).
- Allgemeine Form:
  `FOR` `<Zaehler>` **`FROM`** `<Startwert>` **`TO`** `<Endwert>`
  `BY` `<Schrittweite>` **`DO`** `<Anweisung>` **`ENDDO`**
- Analog zur bedingten Schleife heißt `<Anweisung>` Schleifenrumpf und kann wiederum aus mehreren Anweisungen bestehen.

Bedingte Schleifen in Java

- Java kennt mehrere Arten von bedingten Schleifen.
- Schleifen mit dem Schlüsselwort **`while`**:
  - Die klassische While-Schleife:

```
while (<Bedingung>)
  <Anweisung>
```

  - Die Do-While-Schleife:

```
do
  <Anweisung>
while (<Bedingung>)
```

- Dabei bezeichnet `<Bedingung>` ein Ausdruck vom Typ **`boolean`** und `<Anweisung>` ist entweder eine einzelne Anweisung, oder ein Block mit mehreren Anweisungen.
- Unterschied: `<Anweisung>` wird bevor bzw. nach Überprüfung von `<Bedingung>` ausgeführt.
- Hat `<Bedingung>` den Wert **`false`**, wird die Schleife verlassen.

Bedingte Schleifen in Java

- Eine weitere bedingte Schleife kann in Java mit dem Schlüsselwort **`for`** definiert werden:

```
for (<Initialisierung>; <Bedingung>; <Update>)
  <Anweisung>
```

- Alle drei Bestandteile im Schleifenkopf sind Ausdrücke (nur `<Bedingung>` muss vom Typ **`boolean`** sein).
- Vorsicht: Dieses Konstrukt ist keine klassische gezählte Schleife (auch wenn es **`for`**-Schleife genannt wird).

Bedingte Schleifen in Java

- Die Bestandteile im Einzelnen
  - Der Ausdruck `<Initialisierung>`
    - wird einmal vor dem Start der Schleife aufgerufen
    - darf Variablendeklarationen mit Initialisierung enthalten (um einen Zähler zu erzeugen); diese Variable ist nur im Schleifenkopf und innerhalb der Schleife (`<Anweisung>`) sichtbar
    - darf auch fehlen
  - Der Ausdruck `<Bedingung>`
    - ist ähnlich wie bei den While-Konstrukten die Testbedingung
    - wird am Beginn der Schleife überprüft
    - die Anweisung(en) `<Anweisung>` wird (werden) nur ausgeführt, wenn der Ausdruck `<Bedingung>` den Wert **`true`** hat
    - kann fehlen (gleichbedeutend mit dem Ausdruck **`true`**)
  - Der Ausdruck `<Update>`
    - verändert üblicherweise den Schleifenzähler (falls vorhanden)
    - wird am Ende jedes Schleifendurchlaufs ausgewertet
    - kann fehlen

Laufanweisungen in Java

- Eine gezählte Schleife wird in Java wie folgt mit Hilfe der **`for`**-Schleife notiert:

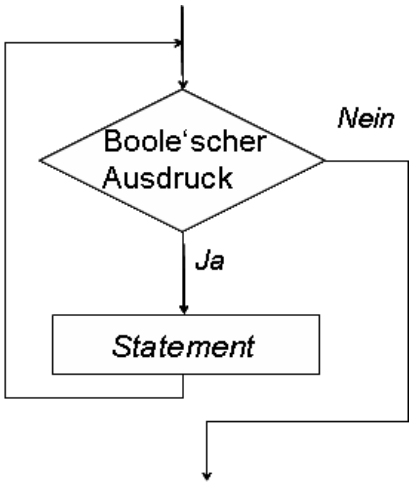
```
for (<Zaehler>=<Startwert>;
     <Zaehler> <= <Endwert>;
     <Zaehler> = <Zaehler> + <Schrittweite>)
  <Anweisung>
```

- Beispiel (Fakultät):

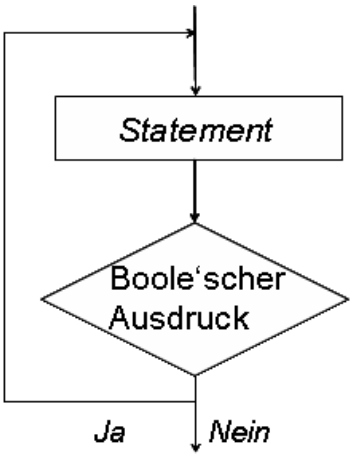
```
public static int fakultaet(int n)
{
  int erg = 1;
  for(int i=1; i<=n; i++)
  {
    erg = erg * i;
  }
  return erg;
}
```


Überblick: Schleifen in Java

`while`-Schleife



`do`-Schleife



Die Befehle `break` und `continue`

- In Java gibt es Möglichkeiten, die normale Auswertungsreihenfolge innerhalb einer **`do`**-, **`while`**- oder **`for`**-Schleife zu verändern.
- Der Befehl **`break`** beendet die Schleife sofort. Das Programm wird mit der ersten Anweisung nach der Schleife fortgesetzt.
- Der Befehl **`continue`** beendet die aktuelle Iteration und beginnt mit der nächsten Iteration, d.h. es wird an den Beginn des Schleifenrumpfes "gesprungen".
- Sind mehrere Schleifen ineinander geschachtelt, so gilt der **`break`**- bzw. **`continue`**-Befehl nur für die aktuelle (innerste) Schleife.

Die Befehle `break` und `continue`

Sprungbefehle:

- Mit einem *Sprungbefehl* kann man an eine beliebige Stelle in einem Programm "springen".
- Die Befehle **`break`** und **`continue`** können in Java auch für (eingeschränkte) Sprungbefehle benutzt werden.
- Der Befehl **`break`** `<`**`label`**`>;` bzw. **`continue`** `<`**`label`**`>;` muss in einem Block stehen, vor dem die Marke `<`**`label`**`>` vereinbart ist. Er bewirkt einen Sprung an das Ende dieses Anweisungsblocks.

Die Befehle `break` und `continue`

Sprungbefehle (cont.):

- Beispiel:

```
int n = 0;
loop1:
for (int i=1; i<=10, i++)
{
  for (int j=1, j<=10, j++)
  {
    n = n + i * j;
    break loop1;
  }
}
System.out.print(n); // n = 1
```

- Der Befehl `break loop1;` erzwingt den Sprung an das Ende der äusseren `for`-Schleife.

Die Befehle `break` und `continue`

Sprungbefehle (cont.):

Anmerkung:

Durch Sprungbefehle (vor allem bei Verwendung von Labeln) wird ein Programm leicht unübersichtlich und Korrektheitsüberprüfung wird schwierig. Wenn möglich, sollten Sprungbefehle vermieden werden.

Unerreichbare Befehle

- *Unerreichbare Befehle* sind Anweisungen, die (u.a.)
  - hinter einer **`break`**- oder **`continue`**-Anweisung liegen, die ohne Bedingung angesprungen werden,
  - in Schleifen stehen, deren Testausdruck zur Compile-Zeit **`false`** ist.
- Solche unerreichbaren Anweisungen sind in Java nicht erlaubt, sie werden vom Compiler nicht akzeptiert.
- Einzige Ausnahme sind Anweisungen hinter der Klausel

  `if` (**`false`**)

  stehen. Diese Anweisungen werden von den meisten Compilern nicht in den Bytecode übernommen, sondern einfach entfernt. Man spricht von *bedingtem Kompilieren*.

Überblick

Beispiel: Suche in einem Array

- Aufgabe:
  - Sei A ein Array der Länge n mit Werten aus $\mathbb{N}$ und $q \in \mathbb{N}$.
  - Suche q in A und gib, falls die Suche erfolgreich ist, die Position i mit $A[i] = q$ aus. Falls die Suche erfolglos ist, gib den Wert -1 aus.
- Einfachste Lösung: *Sequentielle Suche*
  - Durchlaufe A von Position $i = 0, \ldots, n-1$.
  - Falls $A[i] = q$, gib i aus und beende den Durchlauf.
  - Falls q nicht gefunden wird, gib -1 aus.

```
public static int sequentielleSuche(int q, int[] a)
{
  for(int i = 0; i < a.length; i++)
  {
    if(a[i]==q)
    {
      return i;
    }
  }
  return -1;
}
```


Beispiel: Suche in einem Array

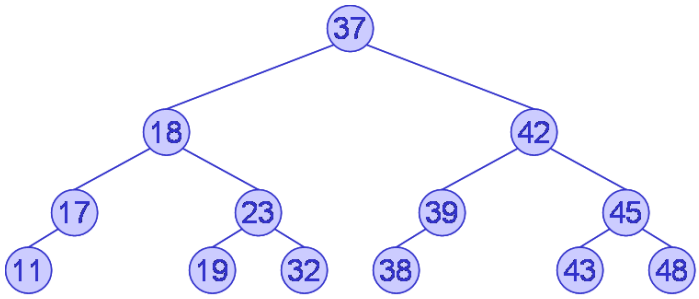
- Anzahl der Vergleiche (= Analyse der Laufzeit):
  - Erfolgreiche Suche: n Vergleiche maximal, $n/2$ im Durchschnitt.
  - Erfolglose Suche: n Vergleiche.
- Falls das Array sortiert ist, funktioniert auch folgende Lösung: *Binäre Suche*
  - Betrachte den Eintrag $A[i]$ mit $i = n/2$ in der Mitte des Arrays
    - Falls $A[i] = q$, gib i aus und beende die Suche.
    - Falls $A[i] > q$, suche in der linken Hälfte von A weiter.
    - Falls $A[i] < q$, suche in der rechten Hälfte von A weiter.
  - In der jeweiligen Hälfte wird ebenfalls mit binärer Suche gesucht.
  - Falls die neue Hälfte des Arrays leer ist, gib -1 aus.

Beispiel: Suche in einem Array

- Beispiel:

A:	11	17	18	19	23	32	37	38	39	42	43	45	48

- Entscheidungsbaum:



- Analyse der Laufzeit: Maximale Anzahl von Vergleichen entspricht Höhe h des Entscheidungsbaums: $h = \lceil \log_2(n+1) \rceil$, also Komplexität in $O(\log n)$.
- Vergleich der Verfahren:
  - $n = 1.000$:
    Sequentielle Suche: 1.000 Vergleiche – Binäre Suche: 10 Vergleiche
  - $n = 1.000.000$:
    Sequentielle Suche: 1.000.000 Vergleiche – Binäre Suche: 20 Vergleiche

Beispiel: Suche in einem Array

Rekursiver Algorithmus in Java:

```
public static int binaereSuche (int q, int[] a, int first, int last)
{
  if (first > last)
  {
    return -1;  // leeres Array
  }
  int m = first + (last-first) / 2;
  if (q == a[m])
  {
    return m;   // q gefunden
  }
  else if (a[m] > q)
  {
    return binaereSuche(q, a, first, m-1);
  }
  else /* a[m] < q */
  {
    return binaereSuche(q, a, m+1, last);
  }
}
public static int binaereSuche (int q, int[] a)
{
  return binaereSuche (q, a, 0, a.length - 1);
}
```

Beispiel: Suche in einem Array

Iterativer Algorithmus in Java:

```
public static int binaereSuche (int q, int[] a)
{
  int first = 0;
  int last = a.length - 1;
  while (last >= first)
  {
    int m = (last + first) / 2;
    if (q == a[m])
    {
      return m;
    }
    else if (q < a[m])
    {
      last = m - 1;
    }
    else /*q > a[m]*/
    {
      first = m + 1;
    }
  }
  return -1; // not found
}
```