

4. Imperative Programmierung

- 4.1 Grunddatentypen und Ausdrücke
- 4.2 Imperative Variablenbehandlung
- 4.3 Anweisungen, Blöcke und Gültigkeitsbereiche
- 4.4 Klassenvariablen
- 4.5 Reihungen
- 4.6 (Statische) Methoden
- 4.7 Kontrollstrukturen
- 4.8 ... putting the pieces together ...

- Das Konzept der Prozedur dient zur Abstraktion von Algorithmen (oder von einzelnen Schritten eines Algorithmus).
- 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.

- Im funktionalen Programmierparadigma werden Algorithmen als Funktionen dargestellt.
- Das imperative Pendant dazu ist die Prozedur, die sogar ein etwas allgemeineres Konzept darstellt: Eine Funktion kann man als Prozedur bezeichnen, aber nicht jede Prozedur ist eine Funktion.
- Eine Funktion stellt nur eine Abbildung von Elementen aus dem Definitionsbereich auf Elemente aus dem Bildbereich dar.
- Es werden aber keine Werte verändert.
- Im imperativen Paradigma können Werte von Variablen verändert werden (durch Anweisungen). Dies kann Effekte auf andere Bereiche eines Programmes haben.
- Treten in einer Prozedur solche sog. Seiteneffekte (oder Nebeneffekte) auf, kann man nicht mehr von einer Funktion sprechen.
- Eine Funktion kann man also als eine Prozedur ohne Seiteneffekte auffassen.

- Wir haben bereits Prozeduren mit Seiteneffekten verwendet:

```
public class HelloWorld
{
    public static final String GRUSS = "Hello, World!";

    public static void main(String[] args)
    {
        System.out.println(GRUSS);
    }
}
```

- Bei einer Funktion ist der Bildbereich eine wichtige Information:

$$f : D \rightarrow B$$

- Bei einer Prozedur, die keine Funktion ist, wählt man als Bildbereich oft die leere Menge:

$$p : D \rightarrow \emptyset$$

Dies signalisiert, dass die Seiteneffekte der Prozedur zur eigentlichen Umsetzung eines Algorithmus gehören, dagegen aber kein (bestimmtes) Element aus dem Bildbereich einer Abbildung als Ergebnis des Algorithmus angesehen werden kann.

- Sehr häufig findet man in imperativen Implementierungen von Algorithmen aber eine Mischform, in der eine Prozedur sowohl Seiteneffekte hat als auch einen nicht-leeren Bildbereich.

- In Java werden Prozeduren durch *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. Dieser Ergebnis-Typ steht für die leere Menge als Bildbereich.
- Eine Methode mit Ergebnistyp `void` 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 mit dem Ergebniswert an, ob der beabsichtigte Nebeneffekt erfolgreich war.
- Das Ergebnis einer Methode ist der Ausdruck nach dem Schlüsselwort `return`. Nach Auswertung dieses Ausdrucks endet die Ausführung der Methode.

```

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);
    }
}

```

Beispiel: Algorithmus für die Funktion

$$f(x) = \left(x + 1 + \frac{1}{x + 1} \right)^2 \quad \text{für } x \neq -1$$

```

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 Wert der Funktion f an der Stelle x
     */
    public static double f(double x)
    {
        double y = x + 1;
        y = y + 1/y;
        y = y * y;
        return y;
    }
}

```

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 ein Methodenaufruf
    }

    public static void main(String[] args)
    {
        // Aufruf der Methode gruessen
        gruessen("Hello, World!"); // abstrakter Parameter gruss
                                   // wird mit konkretem Wert belegt
    }
}
```

- 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.

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!"`

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

- **call-by-value**
Im Aufruf `methodName(parameter)` wird für `parameter` im Methoden-Block eine neue Variable angelegt, in die der Wert von `parameter` geschrieben wird. Auf diese Weise bleibt die ursprüngliche Variable `parameter` von Anweisungen innerhalb der Methode unberührt.
- **call-by-reference**
Im Aufruf `methodName(parameter)` wird die Variable `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.

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?
    }
}
```

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 ?
}
```

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.

4. Imperative Programmierung

- 4.1 Grunddatentypen und Ausdrücke
- 4.2 Imperative Variablenbehandlung
- 4.3 Anweisungen, Blöcke und Gültigkeitsbereiche
- 4.4 Klassenvariablen
- 4.5 Reihungen
- 4.6 (Statische) Methoden
- 4.7 **Kontrollstrukturen**
- 4.8 ... putting the pieces together ...

Motivation:

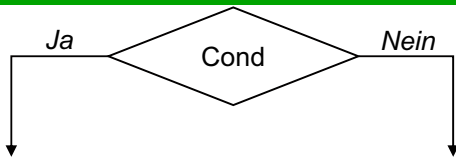
- In vielen Algorithmen benötigt man eine Fallunterscheidung zur Lösung des gegebenen Problems.
- Beispiel: Falls ... dann ... im Algorithmus *Wechselgeld 1*

Führe folgende Schritte der Reihe nach aus:

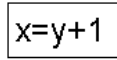
- ① Setze $w = ()$.
- ② Falls die letzte Ziffer von r eine 2,4,7 oder 9 ist, dann erhöhe r um 1 und nimm 1 zu w hinzu.
- ③ Falls die letzte Ziffer von r eine 1 oder 6 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
- ④ Falls die letzte Ziffer von r eine 3 oder 8 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
- ⑤ Solange $r < 100$: Erhöhe r um 5 und nimm 5 zu w hinzu.

- 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.
- Im funktionalen Paradigma modelliert man mit einer Fallunterscheidung eine Abbildung. Deshalb müssen dort im Allgemeinen <Ausdruck1> und <Ausdruck2> den selben Typ haben.
- 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.

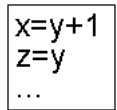
- Man spricht daher im imperativen Paradigma von *bedingten Anweisungen*.
- Die Fallunterscheidung ist ein Spezialfall der bedingten Anweisung.
- Die einfachste Form von bedingten Anweisungen ist:
IF <Bedingung> **THEN** <Anweisungsfolge> **ENDIF**
- Bedingte Anweisungen können beliebig oft verzweigt werden:
IF <Bedingung1> **THEN** <Anweisungsfolge1>
ELSE IF <Bedingung2> **THEN** <Anweisungsfolge2>
:
:
ELSE <AnweisungsfolgeN>
ENDIF
- Die einzelnen Zweige nennt man auch *bewachte Anweisungen*.



Fallunterscheidung bzgl. der Bedingung Cond

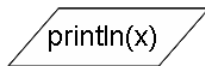


Anweisung



Block

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



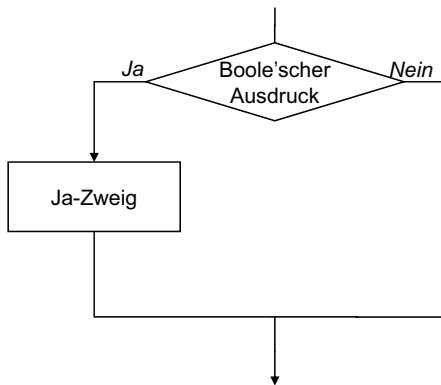
Ein- / Ausgabe



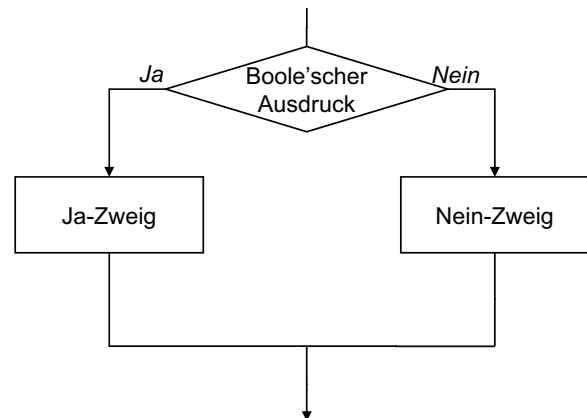
Sequentielle Abfolge

Kontrollflussdiagramme für bedingte Anweisungen

Einfache bedingte Anweisung:



Zweifache bedingte Anweisung:



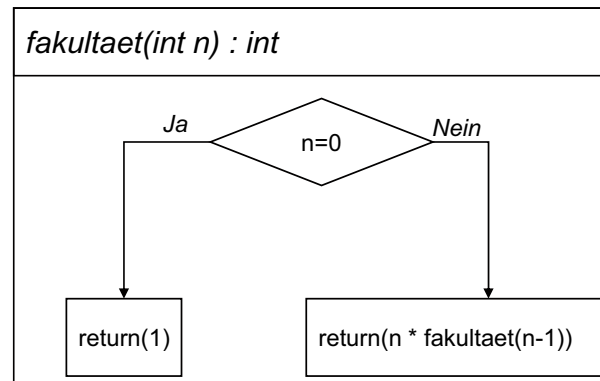
- Rekursion ist ein nützliches und elegantes Entwurfskonzept für Algorithmen.
- Ein rekursiver Entwurf einer Methode verwendet das Ergebnis eines Aufrufs dieser Methode selbst in ihrem Rumpf wieder.
- Damit die Rekursion terminiert, benötigt man in der Regel einen oder mehrere Basisfälle (neben einem oder mehreren Rekursionsfällen).
- Diese verschiedenen Fälle werden typischerweise durch bedingte Anweisungen realisiert.

Eine rekursive Definition in der Mathematik haben wir am Beispiel der Fakultätsfunktion betrachtet:

- Die Fakultäts-Funktion $! : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ ist rekursiv definiert wie folgt:
 - $0! = 1$
 - $(n + 1)! = (n + 1) \cdot (n!)$
- Oft wird äquivalent statt der Rückführung von $n + 1$ auf n der Fall $n \neq 0$ auf $n - 1$ zurückgeführt:

$$n! = \begin{cases} 1, & \text{falls } n = 0, \\ n \cdot (n - 1)!, & \text{sonst.} \end{cases}$$

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



- 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 (also möglicherweise auch einen Block mit mehreren Anweisungen) darstellen.

- 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 fakultaet01(int n)
{
    if (n==0)
    {
        return 1;
    }
    else
    {
        return n * fakultaet01(n-1);
    }
}
```

- Bedingte Anweisungen mit mehr als zwei Zweigen müssen in Java durch Schachtelung mehrerer `if`-Schleifen ausgedrückt werden:

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

:

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

- Gegeben:

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

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

- 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;
    }
}
```

- 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 Zeitraum 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;
```

- 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.

- In Java gibt es eine weitere Möglichkeit, spezielle Mehrfachverzweigungen auszudrücken.
- Die sog. **switch**-Anweisung funktioniert allerdings etwas anders als bedingte Anweisungen.
- Syntax:

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

    :

    default : anweisungN
}
```

- 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.

- Besonderheiten:

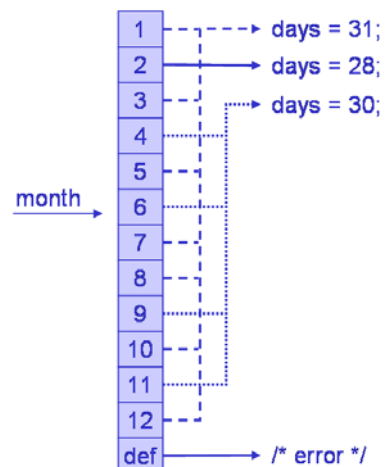
- Der Ausdruck `ausdruck` darf nur vom Typ **byte**, **short**, **int** oder **char** sein.
- Die **case**-Marken sollten alle verschieden sein, müssen aber nicht.
- **Achtung:** Wird zu einer Marke gesprungen, werden alle Anweisungen hinter dieser Marke ausgeführt. Es erfolgt **keine** Unterbrechung, 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.

```

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:



Motivation:

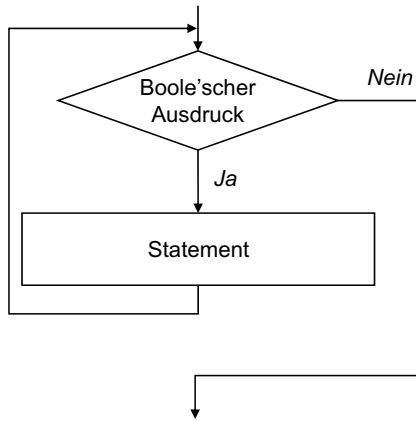
- Im Algorithmus *Wechselgeld 2* hatten wir eine Anweisung der Form **Solange** . . . : . . .

Führe folgende Schritte der Reihe nach aus:

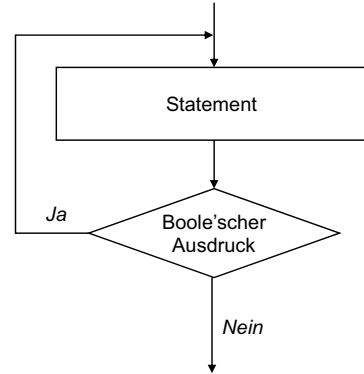
- ① Setze $w = ()$.
- ② Solange $r < 100$: Führe jeweils (wahlweise) einen der folgenden Schritte aus:
 - ① Falls die letzte Ziffer von r eine 2,4,7 oder 9 ist, dann erhöhe r um 1 und nimm 1 zu w hinzu.
 - ② Falls die letzte Ziffer von r eine 1,2,3,6,7 oder 8 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
 - ③ Falls die letzte Ziffer von r eine 0,1,2,3,4 oder 5 ist, dann erhöhe r um 5 und nimm 5 zu w hinzu.

- 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 auch wieder ein Block sein, also aus mehreren Anweisungen bestehen.
- Variante:
DO <Anweisung> **WHILE** <Bedingung> **ENDDO**

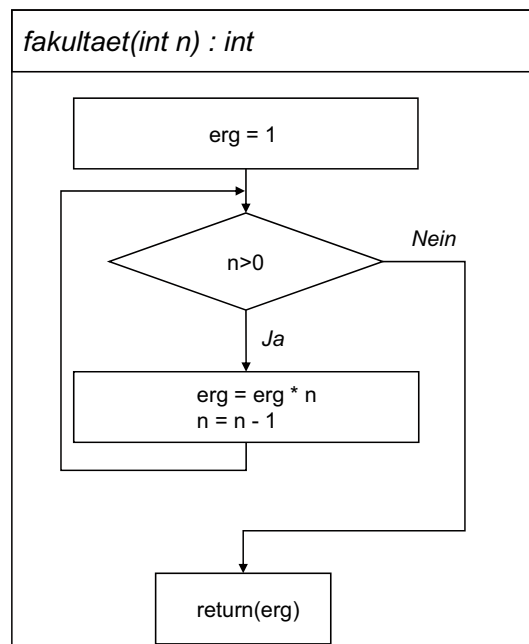
WHILE-Schleife



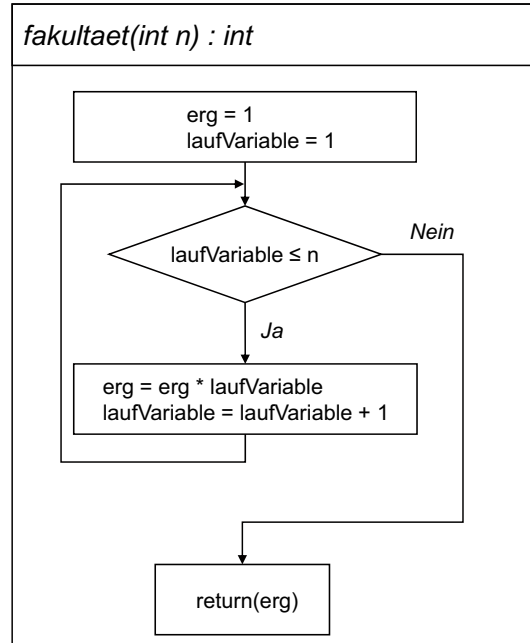
DO-Schleife



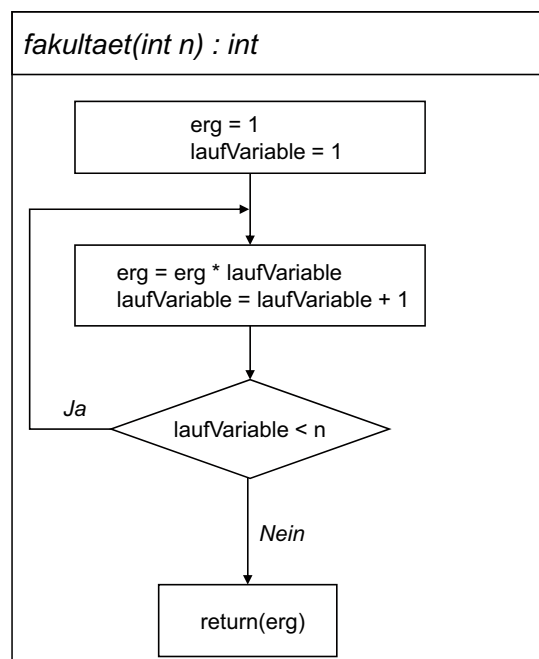
Beispiel: Fakultätsfunktion (nicht rekursiv) mit bedingter Wiederholungsanweisung



Variante: Mitzählen der durchgeführten Schritte



Variante: DO-Schleife



- Die Laufvariable (Zählen der Schritte) wurde in den letzten Beispielen zur Berechnung benutzt.
- Äquivalent zu den Formulierungen der Algorithmen zur Berechnung der Fakultätsfunktion durch die **while**- oder **do**-Schleife kann man das Ergebnis `erg` durch folgende Anweisungsfolge berechnen:

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

- Diese Folge von Zuweisungen könnte man wie folgt imperativ notieren:
Führe für $i = 1, \dots, n$ 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 (bzw. einem Block) bestehen.

- 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 vor bzw. nach der Überprüfung von <Bedingung> ausgeführt.
- Hat <Bedingung> den Wert **false**, wird die Schleife verlassen.

Beispiel (Fakultät):

```
public static int fakultaet02(int n)
{
    int erg = 1;
    while (n > 0)
    {
        erg = erg * n;
        n--;
    }
    return erg;
}
```

Variante: Mitzählen der durchgeführten Schritte

```
public static int fakultaet03(int n)
{
    int erg = 1;
    int laufVariable = 1;
    while (laufVariable <= n)
    {
        erg = erg * laufVariable;
        laufVariable++;
    }
    return erg;
}
```

Variante: DO-Schleife

```
public static int fakultaet04(int n)
{
    int erg = 1;
    int laufVariable = 1;
    do
    {
        erg = erg * laufVariable;
        laufVariable++;
    }
    while (laufVariable < n);
    return erg;
}
```

- 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).

- 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 zu Beginn jedes Schleifendurchlaufs 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

- 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 fakultaet05(int n)  
{  
    int erg = 1;  
    for(int i=1; i<=n; i++)  
    {  
        erg = erg * i;  
    }  
    return erg;  
}
```

- 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.

- 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.

- 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.

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* 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, die 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*.