Datentyp int

Der Datentyp int repräsentiert Ganzzahlen und ist wahrscheinlich der am häufigsten verwendete:

```
int a, b = 0;
a = 10;
```

Das Schöne an Zahlen ist, dass wir sie einfach addieren, subtrahieren, multiplizieren und dividieren können. Aber mit dem Dividieren meinen wir manchmal zwei verschiedene Dinge.

Schauen wir uns ein Beispiel an:

```
int a = 5, b = 2, c;

c = a / b;        // DIV-Operator liefert m
                  // wenn a = m*b+r;

c = a % b;        // MOD-Operator liefert r
```

Wie oft passt b ganzzahlig in a?

Was kleiner als b bleibt übrig?

```
int a = 29, b, c;
b = a / 10;
c = a % 10;
```

Welche Werte haben b und c?

Besondere Betrachtung der Modulo-Operation in Java

Mathematisch gesehen entspricht x modulo y = r der Gleichung:

$$x = y \cdot q + r$$

mit $0 \leq r < y$. Für positive x liefern die mathematische Modulo-Operation und der MOD-Operator das gleiche Resultat. Anders sieht es allerdings bei negativen Eingaben für x aus.

Für negative x haben die Umsetzung des Operators in Java und die mathamatische Modulo-Operation den folgenden Zusammenhang:

x modulo y = x%y + y

Beispiel:

-5 modulo 3 = 1 aber -5%3 = -2

Es gilt:

-5 modulo 3 = 1 = -2+3 = -5%3+3

Im Normalfall werden wir den %-Operator allerdings für positive Zahlen einsetzen. So können wir beispielsweise mit **x%2==0** ermitteln, ob x eine gerade (liefert true) oder ungerade Zahl (liefert false) ist.

Kurzschreibweisen der Standardoperationen I

In Java und anderen Programmiersprachen gibt es für alle Standardoperationen, wie addieren, subtrahieren, usw., eine Kurzschreibweise.

Statt dieser:

<Variable> = <Variable> <Operation> <Ausdruck>;

können wir die meisten zweistelligen Operationen auch kurz so schreiben:

<Variable> <Operation>= <Ausdruck>;

Hier zwei Beispiele dazu:

```
int a = 0, b = 1;
a += b;          // entspricht a = a + b;
b -= 5;          // entspricht b = b - 5;
```

Da solche Standardoperationen typischerweise sehr oft in Programmen vorkommen, erlaubt diese Art der Notation eine Verkürzung der Programme.

Kurzschreibweisen der Standardoperationen II

Für Addition und Subtraktion um 1 (wird auch als inkrementieren und dekrementieren bezeichnet), deren Verwendung wahrscheinlich am häufigsten ist, gibt es sogar noch kürzere Schreibweisen:

```
a++;        // Postinkrement: entspricht a = a + 1
++a;        // Preinkrement:  entspricht a = a + 1
a--;        // Postdekrement: entspricht a = a - 1
--a;        // Predekrement:  entspricht a = a - 1
```

Den kleinen aber feinen Unterschied der Pre- und Postvarianten zeigt folgendes Beispiel:

```
int a, b, c = 0;
a = c++;    // erst Wert von c in a kopieren dann c=c+1
b = ++c;    // c=c+1 dann neuen Wert von c in a kopieren
```

Welche Werte haben a und b?

Kurzschreibweisen der Standardoperationen III

Wenn sich der Kontext allerdings auf die eigene Variable bezieht, dann sollten wir etwas aufpassen, was das folgende Beispiel verdeutlicht:

```
int c = 0;

c = ++c;   // c=c+1 dann neuen Wert von c in c kopieren
System.out.println("c:"+c);

c = c++;   // erst Wert von c in c kopieren dann c=c+1
System.out.println("c:"+c);
```

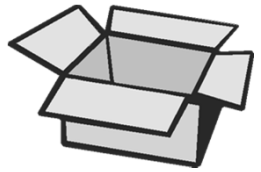
Die Anweisung System.out.println() gibt im Konsolenfenster eine Textzeile aus. Später kommen wir noch einmal intensiv darauf zu sprechen.

Wir würden also erwarten, dass die erste Ausgabe von c eine 1 und die zweite eine 2 liefert. Dem ist aber nicht so:

```
c: 1
c: 1
```

Beim zweiten Fall c=c++ wird zunächst die Zuweisung vermerkt, dann zwar c um 1 kurz erhöht, aber anschließend durch den Vermerk wieder überschrieben.

Datentypen byte, short und long



Die ebenfalls ganzzahligen Datentypen byte, short und long unterscheiden sich dem Datentypen int gegenüber nur in der Größe der zu speichernden Werte:

```
byte b1=0, b2=100;
b1 = b2;
b2 = 100%15;

short s1=4, s2=12;
s1 -= s2;
s2 /= 2;

long a1=48659023, a2=110207203;
a1 *= a2;
```

Ganzzahlige Datentypen lassen sich neben der gebräuchlichen Dezimaldarstellung auch zu den Basen 2 (Binärzahlen), 8 (Oktalzahlen) und 16 (Hexadezimalzahlen) angeben:

```
int zahlBasis10 = 21;      // Dezimaldarstellung
int zahlBasis2  = 0b10101; // Binärdarstellung
int zahlBasis8  = 025;     // Oktaldarstellung
int zahlBasis16 = 0x15;    // Hexadezimaldarstellung
```

$2 \cdot 10^1 + 1 \cdot 10^0$
$1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
$2 \cdot 8^1 + 5 \cdot 8^0$
$1 \cdot 16^1 + 5 \cdot 16^0$

Übersicht bei großen Zahlen behalten

Da große Zahlen manchmal unübersichtlich sein können, ist das Zeichen '_' innerhalb der Darstellung erlaubt. So könnten wir beispielsweise auch schreiben:

```
long a = 48_659_023;
```

Die Trennstriche dienen nur der Visualisierung. Für den späteren Programmablauf bleibt der Inhalt unverändert, wie folgendes Beispiel zeigt:

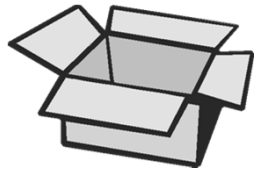
```
long a = 48_659_023;
long b = 4_8_6_5_9_0_2_3;

boolean c = a == b;
System.out.println(c);
```

Wir erhalten:

```
true
```

Datentypen float und double I



Die beiden primitiven Datentypen float und double repräsentieren gebrochene Zahlen (oder Gleitkommazahlen):

```
double a = 2;    // ganze Zahl wird double zugewiesen
double b = 2.2;  // 2.2 wird als double interpretiert

float  c = 3;    // ganze Zahl wird float zugewiesen
float  d = 3.3f; // das f signalisiert den Float

float  e, f;
e  = d%c;
e  = d/c;
f  = c*(d/2.4f) + 1;
f += 1.3f;
```

Gleitkommadarstellung

TODO: Problembeschreibung

```
float  c = 3;    // ganze Zahl wird float zugewiesen
float  d = 3.3f; // das f signalisiert den Float

float  e = d%c;
```

Lösung später noch bringen …

1) SUN: http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.2.3

Datentypen float und double I

Gleitkommazahlen lassen sich auch in der Exponentenschreibweise[1)] angeben, wobei der erste Teil vor dem E (oder e) der Mantisse und der zweite Teil danach dem Exponenten entspricht:

```
double zahl = 1.526552E5;   // entspricht 1.526552 * 10^5
System.out.println("Zahl: " + zahl);

zahl = 4.2E-2;              // entspricht 4.2 * 10^-2
System.out.println("Zahl: " + zahl);
```

Wir erhalten als Ausgabe die beiden Werte:

```
Zahl: 152655.2
Zahl: 0.042
```

1) SUN: http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.2.3

Bindungsstärken der Operatoren

Falls Operatoren die gleiche Bindungsstärke besitzen, werden alle binären Operatoren ausser Zuweisungsoperatoren von links nach rechts ausgewertet. Zuweisungsoperatoren werden von rechts nach links ausgewertet.

Demzufolge ist es auch erlaubt, mehrere Zuweisungen in einer Anweisung zu schreiben:

```
a = b = c = 5;
```

In den Ausdrücken können wir zusätzlich Klammern verwenden, um die Bindungsstärken dem Berechnungswunsch anzupassen. Ein typisches Beispiel ist die Regel „Punkt-vor-Strich", die auch in unserer Tabelle abgebildet ist:

```
a = 1 + 2 * 3;   // a = 7
a = (1 + 2) * 3; // a = 9
```

Im ersten Fall wird die Multiplikation vor der Addition ausgeführt (Bindungsstärke) und im zweiten Fall zuerst der Klammerausdruck.

Bindungsstärken der Operatoren - Übersicht

Operatortyp	Operatoren	Bindungsstärke
postfix	<expr>++, <expr>--	14
prefix, unär, NICHT	++<expr>, --<expr>, +<expr>, -<expr>, ~, !	13
Multiplikation, Division	*, /, %	12
Addition, Subtraktion	+, -	11
Shiften	<<, >>, >>>	10
Relationale, Typerkennung	<, >, <=, >=, instanceof	9
Gleichheit, Ungleichheit	==, !=	8
bitweises UND	&	7
bitweises XOR	^	6
bitweises ODER	|	5
UND	&&	4
OR	||	3
dreistelliger Zuweisungsop.	<expr> ? <value> : <value>	2
Zuweisungsoperatoren	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>>=	1

Inhalte kopieren

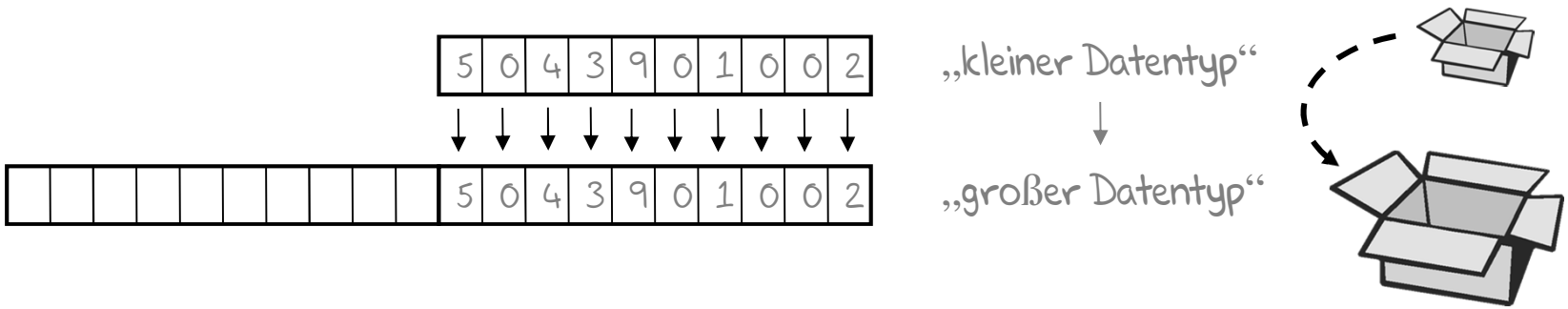
TODO: kein Problem bei gleichen Kisten

```
TODO
```

1) SUN: http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.2.3

Umwandlungen von Datentypen - implizit

Datentyp in größeren kopieren:



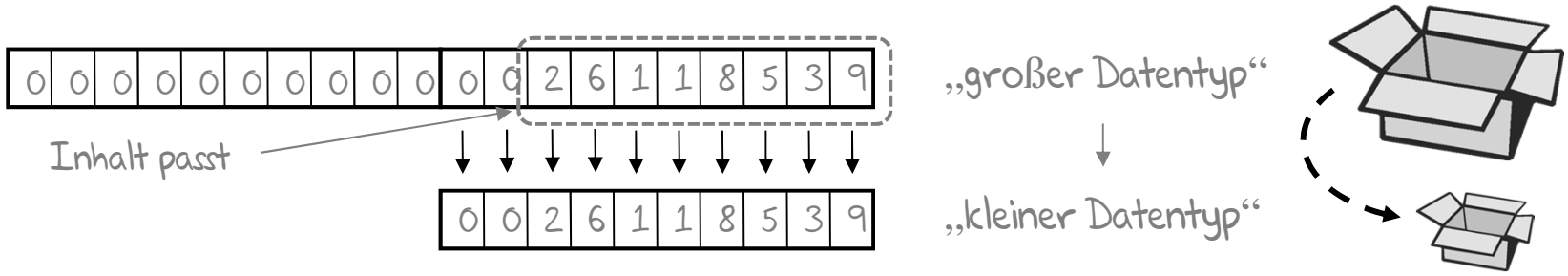
Beispiele:

```
byte a = 28;
int b;
b = a;      // byte (8 Bit) in grösseren int (32 Bit) kopieren

float f = 2.3f;
double d;
d = f;      // float (32 Bit) in grösseren double (64 Bit) kopieren
```

Umwandlungen von Datentypen - explizit

Datentyp in kleineren kopieren:



Fehlerhaftes Beispiel:

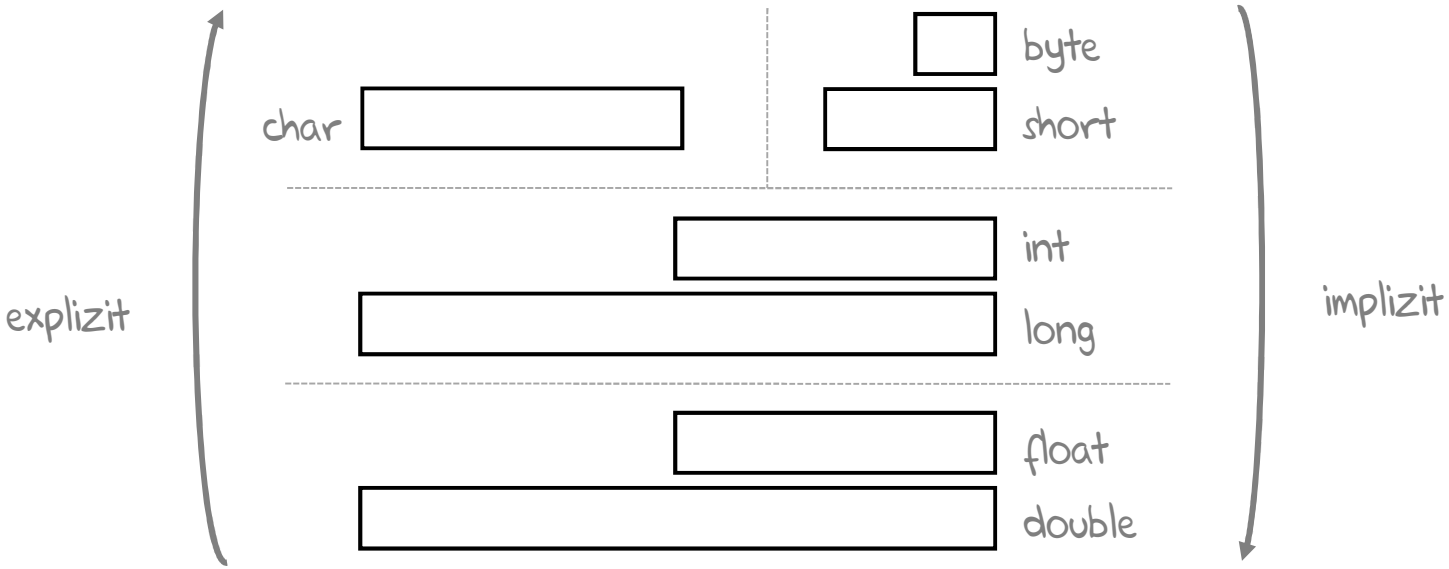
```
float f;
double d = 2.3;
f = d;      // double (64 Bit) in kleineren float (32 Bit) kopieren
```

Wir müssen explizit angeben, dass wir kopieren wollen und uns der Tragweite bewußt sind (Erinnerung an zyklisches Verhalten beim int):

```
float f;
double d = 2.3;
f = (float)d;
```

Übersicht zu impliziten Umwandlungen

Von oben nach unten können wir die erlaubten, impliziten Typumwandlungen ablesen:



So dürfen wir beispielsweise den Inhalt eines byte ohne zusätzliche Angabe in einen int kopieren. Auch ein char darf implizit in einen int umgewandelt werden, aber nicht umgekehrt.

Die Größen der Kästchen entsprechen dem relativen Speicherbedarf. Von unten nach oben müssen die Datentypen explizit umgewandelt werden.

Implizite Umwandlungen sind nicht immer verlustfrei

Da stellt sich aber die Frage, wie der Inhalt eines long (64 Bit) ohne Datenverlust in einen float (32 Bit) kopiert werden kann. Das geht natürlich nicht immer.

Sogar bei der Umwandlung eines long in einen double kann Datenverlust erstehen, was das folgende Beispiel zeigt:

```
long l = 10000000000000001L;
double d = l;
System.out.println("Differenz: " + (l - (long)d));
```

Als Ergebnis für die Differenz aus dem ursprünglichen **long** und dem zunächst in einen **double** umgewandelten und dann zurückgecasteten erhalten wir:

```
Differenz: 1
```

Datentypen sind für die Operationen entscheidend

An dieser Stelle ist es wichtig, auf einen häufig gemachten Fehler hinzuweisen:

```
int a=5, b=7;
double erg = a/b;
```

Wir erwarten eigentlich einen Wert um 0.71, erhalten aber 0.0. Was ist der Grund?

Folgende Programmzeilen liefern das gewünschte Ergebnis:

```
int a=5, b=7;
double erg = (double)a/b;
```

Vorlesungsteil

Grundlegende Prinzipien der Programmentwicklung

Nur Persönlichkeiten bewegen die Welt, niemals Prinzipien.
Oscar Wilde

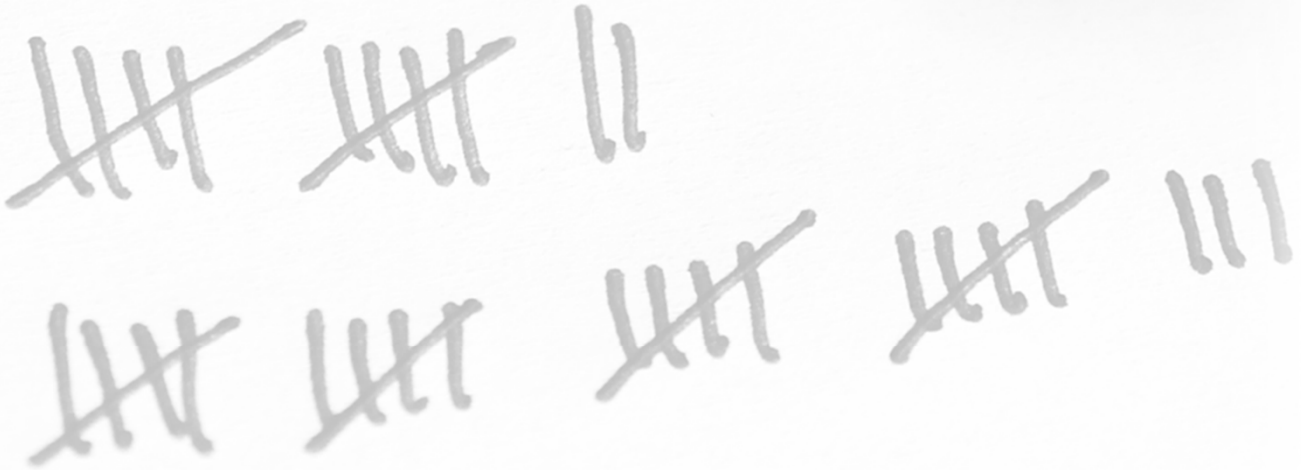
Übersicht zum Vorlesungsinhalt

zeitliche Abfolge und Inhalte können variieren

Grundlegende Prinzipien der Programmentwicklung

- Programm als Kochrezept
- Aktivitätsdiagramm (UML)
- Prinzipien der Programmerstellung
- Pseudocode
- Erstellen eines Javaprogramms
- Einfaches Klassenkonzept
- Kommentare, Syntaxhighlighting, Einrücken
- Sequentielle Anweisungen
- Verzweigungen
- Schleifentypen
- Sprunganweisungen
- Wiederholende Programmabschnitte

Wer von Euch kann Programmieren?



Ihr ALLE könnt es bereits!

Rezept für sechs Portionen von Omas Eierpfannkuchen

Zutaten:	Zubereitung:
4 Eier	- vier Eier in eine Schüssel schlagen und verrühren
Mehl	- solange Mehl hinzugeben, bis Teig schwer zu rühren ist
Milch	- solange Milch hinzugeben, bis die Masse wieder leicht zu rühren ist, damit sie sich gut in der Pfanne verteilen lässt
1 Apfel	
Wurst/Käse	- etwas Fett in einer Pfanne erhitzen
Marmelade	- Teil der Masse in die Mitte der Pfanne geben, bis der Boden bedeckt ist
Apfelmus	- bei der süßen Variante können Apfelscheiben dazu gegeben werden, bei der herzhaften Variante Wurst und Käse
Zucker	
	- die Eierpfannkuchen von beiden Seiten gut anbraten
	- die süßen Eierpfannkuchen können nach belieben, z.B. mit Marmelade, Apfelmus oder Zucker garniert werden

Die Reihenfolge der Anweisungen ist dabei wichtig!

Pseudocode vom Rezept

Mit Pseudocode bezeichnen wir die lesbare Notation eines Programms in keiner spezifischen Programmiersprache, sondern in einer allgemeineren Form:

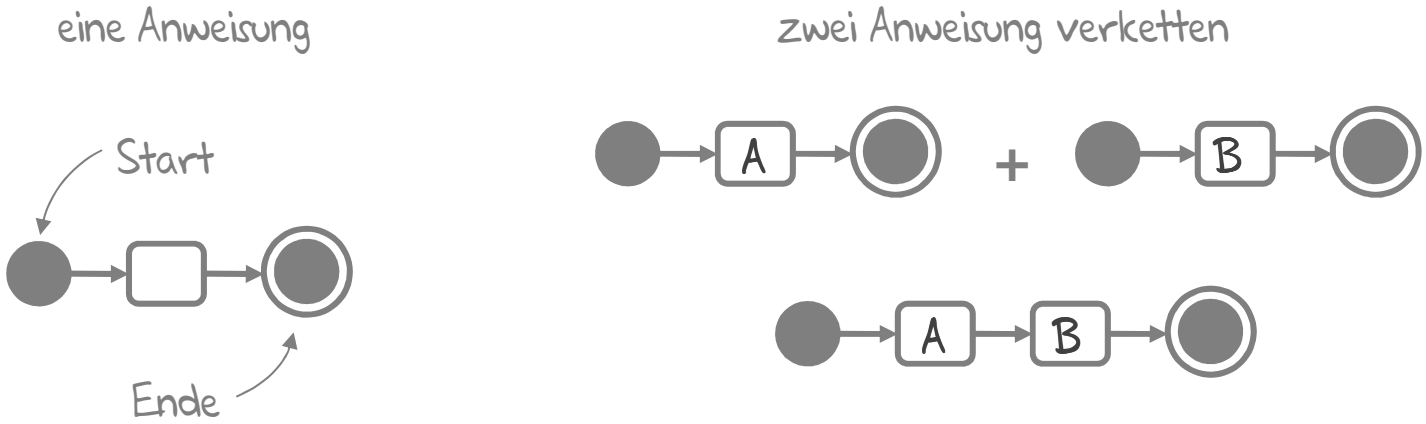
```
 1 vier Eier in eine Schüssel schlagen und verrühren
 2 Mehl in die Schüssel hinzugeben
 3  -> wenn Teig noch nicht schwer zu rühren ist gehe zu 2
 4 Milch in die Schüssel geben
 5  -> wenn Teig nicht leicht zu rühren ist gehe zu 4
 6 etwas Fett in einer Pfanne erhitzen
 7 einen Teil in die Pfanne geben, bis Boden gut bedeckt
 8  -> wenn süße Variante gewünscht gehe zu 9 ansonsten zu 10
 9 Apfelscheiben hinzugeben, gehe zu 11
10 Wurst und Käse hinzugeben
11 die Eierpfannkuchen von beiden Seiten gut braun braten
12  -> wenn süße Variante gewünscht gehe zu 13 ansonsten zu 14
13 mit Marmelade, Apfelmus oder Zucker garnieren
14 FERTIG
```

Alltägliche Prozesse können als Programme verstanden werden. Es gibt immer wiederkehrende Vorgehensweisen. Bei genauerer Betrachtung gibt es sogar nur drei.

Alle anderen Methoden sind leicht als Spezialfall dieser drei zu interpretieren.

Aktivitätsdiagramm

Ein weißes Kästchen steht für eine Anweisung, die das Programm auszuführen hat:



Eine Anweisung mit der gestartet wird, markieren wir zusätzlich mit einem ausgefüllten Kreis. Eine finale Anweisung wird ebenfalls gesondert markiert.

Diese Abbildungsform wird auch als Aktivitätsdiagramm bezeichnet und gehört zum Sprachumfang der grafischen Modellierungssprache UML (Unified Modeling Language) die aktuell in der Softwareentwicklung standardmäßig eingesetzt wird.

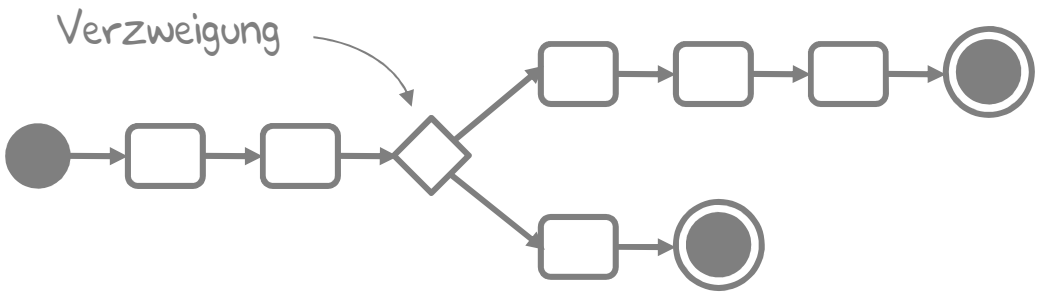
Sequentieller Programmablauf

Prinzipiell kann ein Programm Anweisung für Anweisung hintereinander weg geschrieben werden. Kein Abschnitt wird dabei wiederholt:

Das könnte beispielsweise eine maschinelle Qualitätsprüfung sein (Prüfen: Größe, Form, Farbe, ...)

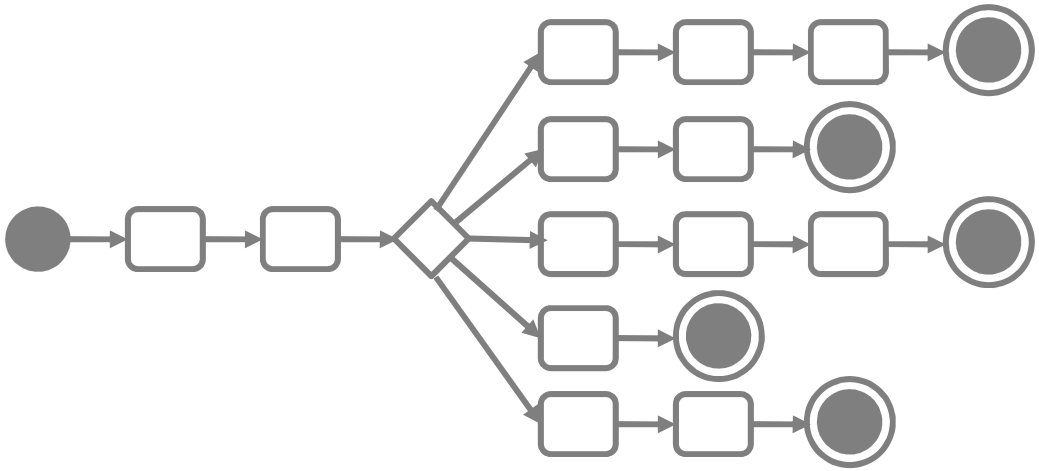
Verzweigungen

Oft kommt es vor, dass eine Entscheidung getroffen werden muss, die die Wahl der nachfolgenden Anweisungen beeinflusst:



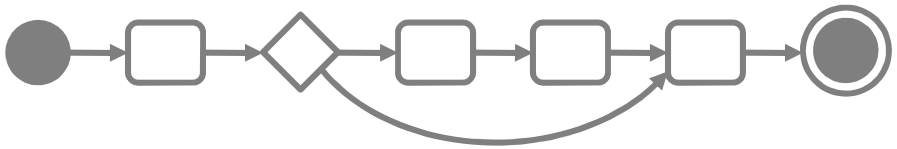
In unserem Rezeptbeispiel ist es beispielsweise die Wahl der Variante.

Es gibt auch die Möglichkeit, eine Verzweigung in beliebig vielen Wegen fortzufahren. Wir sprechen dann von einer Mehrfachverzweigung:



Sprünge

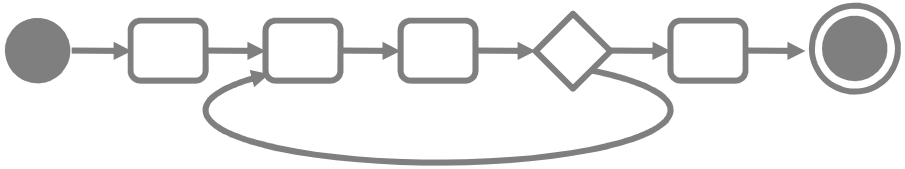
Sprünge sind ein Spezialfall einer Verzweigung:



Anstatt einen unabhängigen Weg zu beschreiten, springen wir zu einen späteren Programmabschnitt und machen dort weiter. Damit können wir Programmpassagen bei Bedarf überspringen.

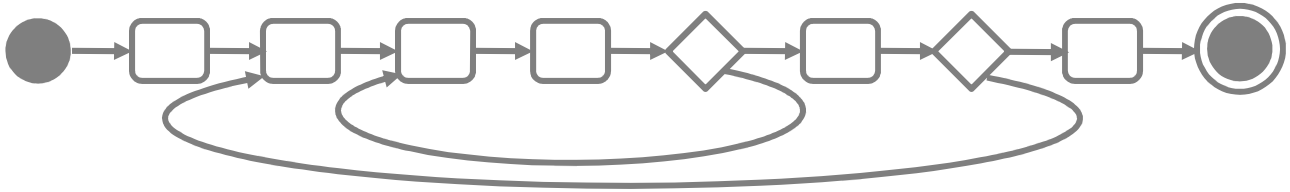
Schleifen

Schleifen sind ebenfalls ein Spezialfall einer Verzweigung:



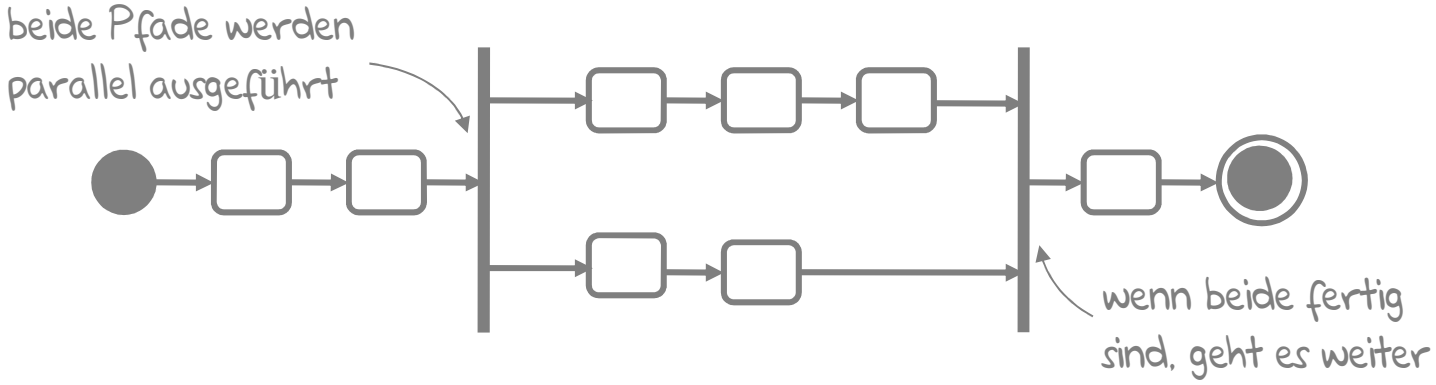
Wir können einen bestimmten Prozessabschnitt solange wiederholen lassen (z.B. Mehl in eine Schüssel geben) bis ein gewünschtes Ergebnis erreicht ist.

Schleifen kommen oft verschachtelt vor (Mehrfachschleifen):



Parallelität

Das dritte wichtige Konzept, neben sequentieller Abfolge und dem Sprung ist die Parallelität:



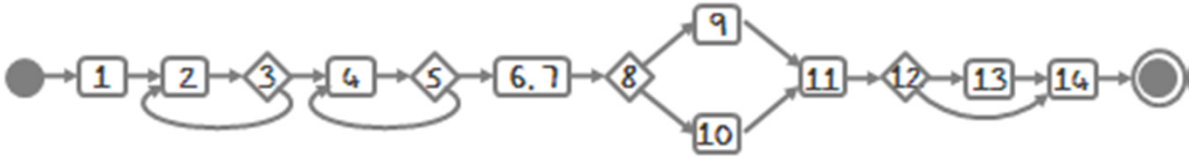
Zwei Striche auf den Verzweigungspfeilen sollen bedeuten, dass die beiden Wege gleichzeitig bearbeitet werden sollen und sich später wieder treffen können.

Kombination zu Programmen

In der Kombination ergeben die vorgestellten Methoden: Programme! Schauen wir uns dazu nochmal das Kochrezeptbeispiel an:

```
 1 vier Eier in eine Schüssel schlagen und verrühren
 2 Mehl in die Schüssel hinzugeben
 3  -> wenn Teig noch nicht schwer zu rühren ist gehe zu 2
 4 Milch in die Schüssel geben
 5  -> wenn Teig nicht leicht zu rühren ist gehe zu 4
 6 etwas Fett in einer Pfanne erhitzen
 7 einen Teil in die Pfanne geben, bis Boden gut bedeckt
 8  -> wenn süße Variante gewünscht gehe zu 9 ansonsten zu 10
 9 Apfelscheiben hinzugeben, gehe zu 11
10 Wurst und Käse hinzugeben
11 die Eierpfannkuchen von beiden Seiten gut braun braten
12  -> wenn süße Variante gewünscht gehe zu 13 ansonsten zu 14
13 mit Marmelade, Apfelmus oder Zucker garnieren
14 FERTIG
```

Dargestellt als Aktivitätsdiagramm:



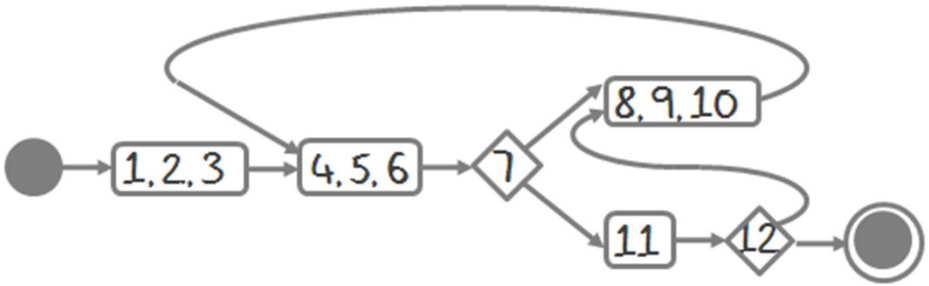
Wir erlauben mehrere nacheinanderfolgende Anweisungen in einem Kästchen zu notieren, wenn es der kompakten Übersicht dienlich ist.

Erstellen eines Javaprogramms in Pseudocode

Wir können mal an dieser Stelle die Erstellung eines Javaprogramms mit Hilfe von Pseudocode ausdrücken:

```
 1 öffne einen Texteditor
 2 lege ein Dokument mit gewünschtem Programmnamen an
 3 schreibe ein Javaprogramm
 4 speichere es mit der Endung ".java" in Ordner <X>
 5 gehe außerhalb des Editors in einer Konsole zu Ort <X>
 6 schreibe "javac <Programmname>.java"
 7  -> wenn der Javacompiler Fehler ausgibt gehe zu 8 sonst 11
 8 gehe zurück zum Editor
 9 korrigiere angezeigte Fehler
10 gehe zu 4
11 schreibe "java <Programmname>"
12 -> wenn das Programm noch nicht das Gewünschte tut gehe zu 8
13 FERTIG
```

Auch das können wir wieder als Aktivitätsdiagramm darstellen:



Erstellen eines Javaprogramms

Wir können Javaprogramme mit folgender Syntax in der Konsole kompilieren:

javac <Programmname>.java

und anschließend so ausführen:

java <Programmname>

Versuchen wir an dieser Stelle einmal die folgenden Programmzeilen entsprechend den Anweisungen des Pseudocodes einzugeben und das Programm zu starten.

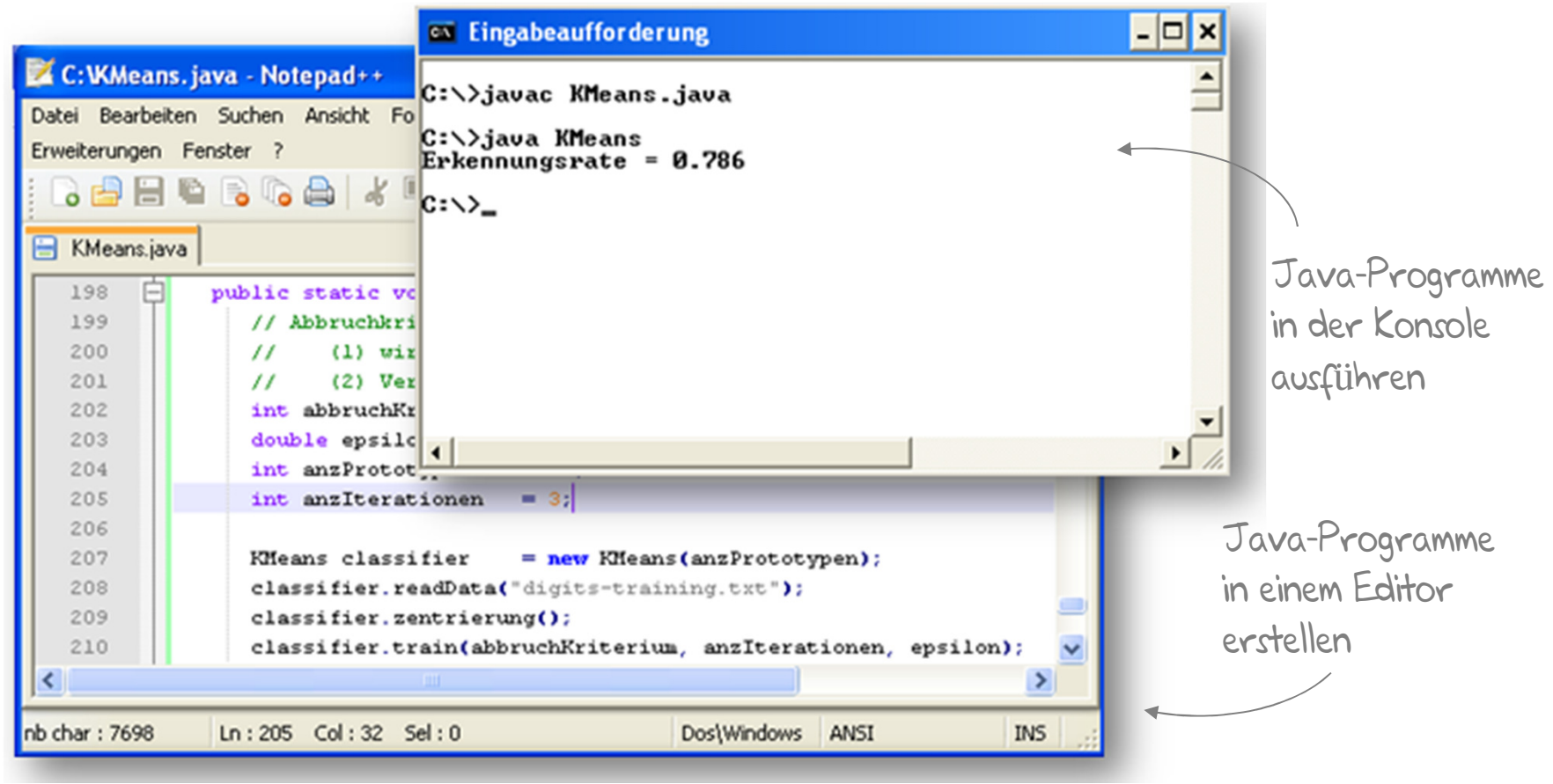
```
public class TestProgramm {
   public static void main(String[] args) {
      System.out.println("Das habe ich ganz allein geschafft!");
   }
}
```

Wenn wir alles richtig gemacht haben, wird eine Textzeile in der Konsole ausgegeben:

```
C:\Java>javac TestProgramm.java
C:\Java>java TestProgramm
Das habe ich ganz allein geschafft!
```

Editor und Konsole

Kompilieren und Ausführen von Programmen:



Programmieren mit einem einfachen Klassenkonzept

Das folgende Programm tut erstmal herzlich wenig, soll aber zeigen, welche Zeilen wir für alle folgenden Beispiele benötigen:

```
public class MeinErstesProgramm {
   public static void main(String[] args) {
      // HIER KOMMEN DIE ANWEISUNGEN DES PROGRAMMS HIN
   }
}
```

Das Einrücken der Zeilen innerhalb eines Blocks (so wird der Abschnitt zwischen { und } genannt) dient der Lesbarkeit.

Programme kommentieren

Es gibt zwei Möglichkeiten in Java Kommentare zu schreiben:

```
// Ich bin ein hilfreicher Kommentar in einer Zeile

/* Falls ich einen Kommentar über mehrere Zeilen
   hinweg schreiben möchte, so verwende ich ein
   öffnendes und schließendes Kommentarsymbol
*/

public class Kommentierung {     // ich kann auch hier stehen
   public static void main(String[] args) {

      /* An diese Stelle schreiben wir die
         Programmanweisungen */

   }
}
```

Wir müssen immer darauf achten, unsere Programme nicht nur ausreichend, sondern verständlich zu kommentieren. Es ist nicht immer einfach, ein langes Programm ohne hilfreiche Kommentare zu verstehen. Dies trifft sogar auf selbst geschriebene Programme zu, erst recht auf die von anderen.

Daher sind gerade in Projekten, bei denen mehrere Programmierer zusammenarbeiten, Kommentare unverzichtbar.

Programme kommentieren – Variante I

Je besser das Programm geschrieben ist, desto weniger Kommentare sind notwendig.

```
public static boolean funktion(String s) {
   int i = s.toCharArray().length/2;
   for (char c:s.toCharArray()) {
      if (s.toCharArray()[i] != s.toCharArray()[s.toCharArray().length-i-1])
         return false;
      i--;
      if (i==0) break;
   }
   return true;
}
```

```
System.out.println(args[0]+" ist "+(funktion(args[0])?"":"k")+"ein Palindrom");
```

Programme kommentieren – Variante II

```
public static boolean istPalindrom(String text) {
   char[] zeichen = text.toCharArray();
   int textMitte  = text.length()/2;
   boolean istPalindrom = true;

   for (int i=0; i<textMitte; i++) {
      char zeichenVorn   = zeichen[i];
      char zeichenHinten = zeichen[zeichen.length-i-1];

      if (zeichenVorn != zeichenHinten) {
         istPalindrom = false;
         break;
      }
   }
   return istPalindrom;
}
```

```
String programmEingabe = args[0];
if (istPalindrom(programmEingabe))
   System.out.println(programmEingabe+" ist ein Palindrom");
else
   System.out.println(programmEingabe+" ist kein Palindrom");
```

Einrücken von Anweisungsblöcken

Kommentare und Syntaxhighlighting reichen nicht!

```
public static int zaehleUmgebung(boolean[][] m, int x, int y){
int ret = 0;
for (int i=(x-1);i<(x+2);++i){
for (int j=(y-1);j<(y+2);++j){
try {if (m[i][j]) ret += 1;}catch (IndexOutOfBoundsException e){}
}
}// einen zuviel mitgezaehlt?
if (m[x][y])ret -= 1;
return ret;
}
```

```
public static int zaehleUmgebung(boolean[][] m, int x, int y) {
   int ret = 0;
   for (int i=(x-1); i<(x+2); ++i) {
      for (int j=(y-1); j<(y+2); ++j) {
         try {
            if (m[i][j])
               ret += 1;
         } catch (IndexOutOfBoundsException e) {}
      }
   }

   // einen zuviel mitgezaehlt?
   if (m[x][y])
      ret -= 1;

   return ret;
}
```

Sequentielle Anweisungen

Wir haben bereits gesehen, dass Anweisungen mit ; abgeschlossen werden:

```
public class Sequentiell1 {
   public static void main(String[] args) {
      int a=5;     // Anweisung 1
      a=a*2;       // Anweisung 2
      a=a+10;      // Anweisung 3
      a=a-5;       // Anweisung 4
   }
}
```

Welchen Wert hat a?

Wir können das überprüfen: Geben wir dazu am Ende noch folgende Zeile dazu:

```
System.out.println("a hat den Wert: "+a);
```

Live-Coding-Session

Verzweigungen mit if I

Die Syntax für eine Verzweigung kann auf verschiedene Weisen formuliert werden.

Wenn eine Bedingung erfüllt ist, führe eine einzelne Anweisung aus. Die Syntax dafür sieht wie folgt aus:

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

Ein Beispiel:

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

Welchen Wert repräsentiert a?

Es können auch mehrere Anweisungen (ein Anweisungsblock) ausgeführt werden:

```
if (<Bedingung>) {
   <Anweisung1>;
   <Anweisung2>;
   ...
}
```

Verzweigungen mit if II

Eine Erweiterung dazu ist die if-else-Verzweigung:

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

Ein Beispiel:

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

Wir können auch Bedingungen verknüpfen:

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

Guter Programmierstil

Da bei einer Bedingung immer ein Ausdruck steht, der entweder true oder false ist, können wir den folgenden Abschnitt:

```
if (a == true)
   // tue etwas
```

ersetzen durch:

```
if (a)
   // ist a true, tue etwas
```

Der Grund ist einleuchtend, da bei jedem Wert von a der Ausdruck a==(a==true) erfüllt ist.

Wird a auf false getestet, machen wir das entsprechend so:

```
if (!a)
   // ist a false, tue etwas
```

Verzweigung mit switch I

Um nun Mehrfachverzweigungen zu realisieren und nicht zu einer unübersichtlichen Flut von if-Verzweigungen greifen zu müssen, steht uns switch zur Verfügung:

```
switch (<Ausdruck>) {
  case <Konstante1>:
      <Anweisung1>;
      [break;]

  case <Konstante2>:
      <Anweisung2>;
      [break;]

 [default:]
      <Anweisung3>;
}
```

Leider gibt es für die Verwendung ein paar Einschränkungen, so können wir nur Bedingungen überprüfen in der Form: *Hat int a den Inhalt 4?*

Als Ausdruck lassen sich verschiedene primitive Datentypen auf Ihren Inhalt untersuchen. Zu den erlaubten gehören:
char, byte, short, int und (seit Java 7 auch) String.

Verzweigung mit switch II

Hier ein kleines Beispiel zu switch:

```
int i = 0;    // hier mal unterschiedliche i probieren
switch(i){
   case 0:
      System.out.println("0");
      break;
   case 1:
      System.out.println("1");
   case 2:
      System.out.println("1 oder 2");
      break;
   case 3:
      System.out.println("3");
      break;
   default:
      System.out.println("hier landen alle anderen...");
}
```

Wenn wir die switch-Verzweigung beispielsweise mit den Zahlen i=0,1,2,3,4 nacheinander durchlaufen würden, erhielten wir die folgende Ausgabe:

```
C:\Java>java Verzweigung
0
1
1 oder 2
1 oder 2
3
hier landen alle anderen...
```

Verzweigung mit switch III

Variante mit Zeichenketten:

```
String code = "007";
switch(code) {
   case "006":
      System.out.println("Nigel Boswell");
      break;
   case "007":
      System.out.println("James Bond");
      break;
   default:
      System.out.println("kein berühmter Spion");
}
```

Motivation zu Schleifen

Folgendes Programm ist gegeben, das eine Zahl quadriert und das Ergebnis ausgibt:

```
System.out.println("1 zum Quadrat ist "+(1*1));
System.out.println("2 zum Quadrat ist "+(2*2));
System.out.println("3 zum Quadrat ist "+(3*3));
System.out.println("4 zum Quadrat ist "+(4*4));
System.out.println("5 zum Quadrat ist "+(5*5));
System.out.println("6 zum Quadrat ist "+(6*6));
```

Es wäre sicherlich auch eine unangenehme Aufgabe, alle ganzen Zahlen zwischen 1 und 1000 auf diese Weise quadrieren und ausgeben zu lassen. Daher ist das Schleifen-Konzept ziemlich nützlich.

Angenommen, wir möchten die Aufgabe lösen, alle Quadrate der Zahlen zwischen 1 und 1000 auszugeben. Dann könnten wir das in Pseudocode in etwa so ausdrücken:

```
Beginne mit i=1
Gib den Wert i*i aus
Falls i<=1000
    -> Erhöhe i um 1 und springe zu Zeile 2
```

Oder in Worten ausgedrückt: Starte mit i=1 und solange i<=1000 erfüllt ist, mache folgendes: gib das Quadrat von i aus und erhöhe anschließend i um 1.

Schleife mit for

Wir benötigen zunächst eine Variable, die zu Beginn mit einem Startwert initialisiert wird. Die Schleife führt den nachfolgenden Anweisungsblock solange aus, erhöht oder verringert dabei die Variable um einen Wert, bis die angegebene Bedingung nicht mehr erfüllt ist:

```
for (<Initialisierungen>; <Bedingungen>; <Aktualisierungen>) {
   <Anweisung>;
}
```

Im Programm könnten wir die zu Beginn geschilderte Aufgabe, alle Quadratzahlen für die Werte 1 bis 1000 auszugeben, mit einer for-Schleife so lösen:

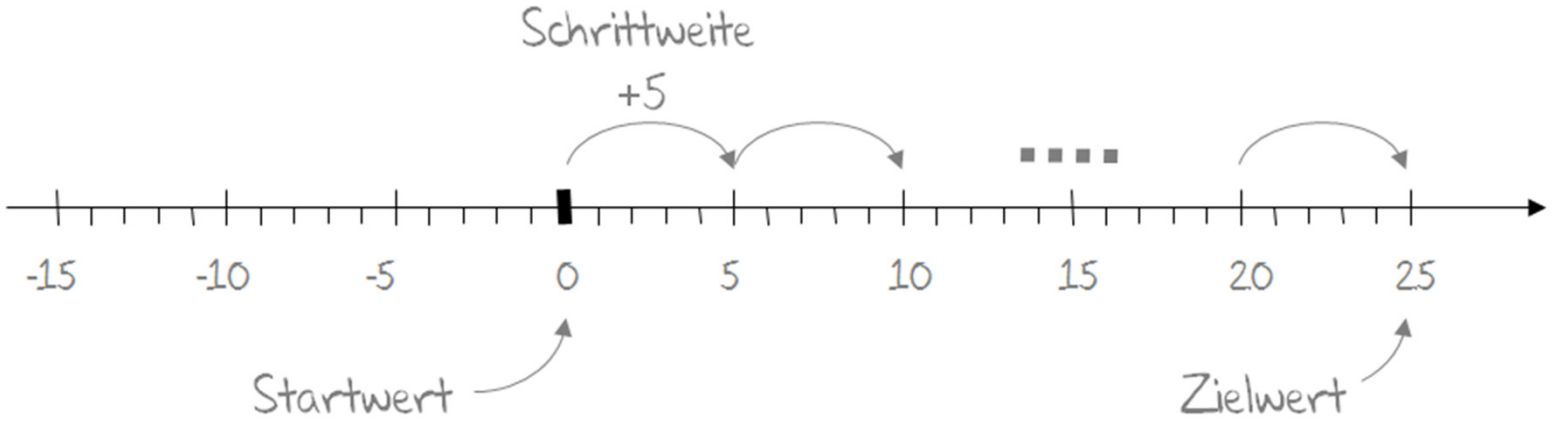
```
for (int i=1; i<=1000; i=i+1)
   System.out.println(i+" zum Quadrat ist "+(i*i));
```

Mit int i=1 geben wir einen Startwert vor. Die Schleife führt die Anweisungen innerhalb der geschweiften Klammern solange aus und erhöht jedes Mal i um 1 bis die Bedingung i<=1000 nicht mehr erfüllt ist.

Beispiele zur for-Schleife I

Wir können aber auch andere Konstruktionen mit einer for-Schleife realisieren.

Beispielsweise könnten wir bei 0 anfangen, in Fünferschritten weiterlaufen und alle Werte bis einschließlich 25 erzeugen:

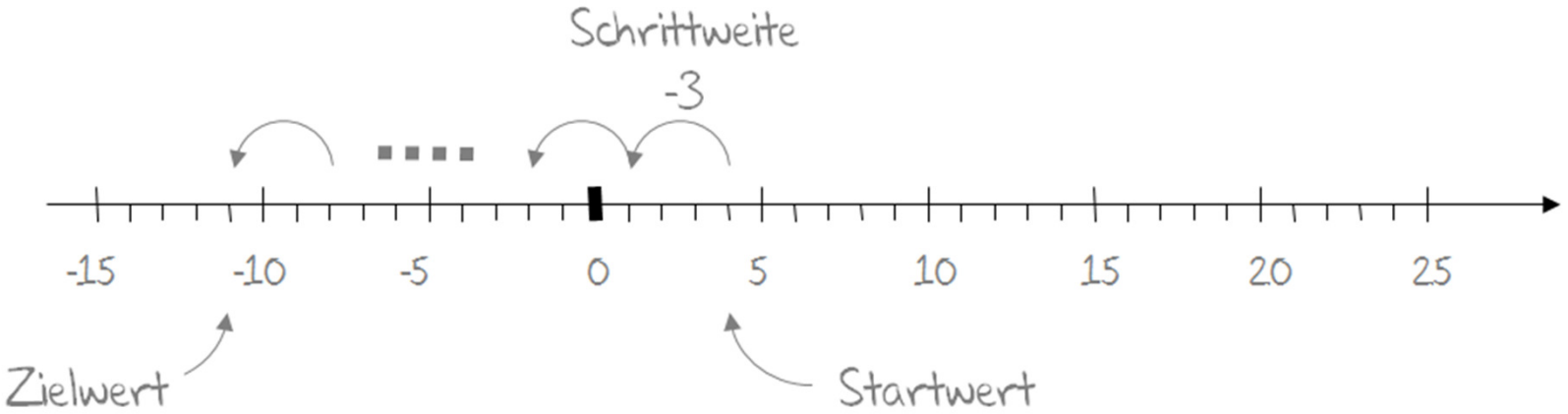


Umsetzung in Java:

```
for (int i=0; i<=25; i=i+5)
   System.out.println("Aktueller Wert für i ist "+i);
```

Beispiele zur for-Schleife II

Eine Schleife muss Variablen nicht immer vergrößern, wir können auch bei 4 beginnend immer 3 abziehen, bis wir bei -11 angelangt sind:



Umsetzung in Java:

```
for (int i=4; i>=-11; i=i-3)
   System.out.println("Aktueller Wert für i ist "+i);
```

Beispiele zur for-Schleife III

Im folgenden Beispiel wollen wir mit einer Schleife die Summe der Zahlen

$$summe(n) = \sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n$$

berechnen:

```
int summe=0;
for (int i=1; i<=n; i++)
   summe += i;
```

Da bei der Definition einer for-Schleife, die drei Teile <Initialisierung>, <Bedingung> und <Aktualisierung> nicht nur optional sind, sondern sogar mehrfach eingesetzt werden dürfen, können wir die Summe sogar ganz ohne Anweisungsteil berechnen:

```
int summe=0;
for (int i=1; i<=n; summe+=i, i++);
```

Soviel an dieser Stelle: Neben der for-Schleife gibt es noch mehr Schleifenvarianten. Jede hat auf Grund der Programmästhetik ihre Existenzberechtigung, obwohl leicht zu zeigen ist, dass wir mit einer Variante immer auskommen könnten.

Schleife mit while

Manchmal ist es nicht klar, wie viele Schleifendurchläufe benötigt werden, um ein Ergebnis zu erhalten. Da wäre es schön, eine Möglichkeit zu haben, wie diese: **Wiederhole die Anweisungen solange, eine Bedingung erfüllt ist.**

Genau diesen Schleifentyp repräsentiert die while-Schleife. Hier die zugehörige Syntax:

```
while (<Bedingung>) {
    <Anweisung>;
}
```

Wie könnten wir die Quadrate von 1 bis 1000 mit while lösen?

Wir werden das Beispiel aus der for-Schleife wieder aufnehmen und sehen, wie das mit while gelöst werden kann:

```
int i=1;
while (i<=1000){
   System.out.println(i+" zum Quadrat ist "+(i*i));
   i=i+1;
}
```

Schleife mit while - Gefahr Endlosschleife

Die Schleife wird einfach solange ausgeführt, bis die Bedingung hinter while nicht mehr erfüllt ist.

Hier ist natürlich auch die Gefahr gegeben, dass die Schleife endlos laufen kann, wie in diesem Beispiel zu sehen ist:

```
int i=1;
while (i<=1000){
   System.out.println(i+" zum Quadrat ist "+(i*i));
}
```

Warum ergibt das eine Endlosschleife?

Schleife mit do-while I

Die do-while-Schleife führt im Gegensatz zur while-Schleife zuerst den Anweisungsblock einmal aus und prüft anschließend die Bedingung:

```
do {
    <Anweisung>;
} while (<Bedingung>);
```

Damit haben wir im Gegensatz zur while-Schleife, die kopfgesteuert ist, eine fußgesteuerte Schleife.

Schauen wir uns dazu mal folgendes Beispiel an:

```
int i=0;
do {
   i++;
   System.out.println("Wert von i: "+i);
} while (i<5);
```

Wir erhalten folgende Ausgabe:

```
C:\Java>java Schleifen
Wert von i: 1
Wert von i: 2
Wert von i: 3
Wert von i: 4
Wert von i: 5
```

Schleife mit do-while II

Wenn wir die Bedingung so ändern, dass sie von vornherein nicht erfüllt ist:

```
int i=0;
do {
   i++;
   System.out.println("Wert von i: "+i);
} while (i<0);
```

geschieht trotzdem das Folgende:

```
C:\Java>java Schleifen
Wert von i: 1
```

Das war auch zu erwarten, da die Überprüfung der Bedingung erst nach der ersten Ausführung stattfindet und deshalb der Schleifeninhalt mindestens einmal ausgeführt wird.