Kleine Java-Einführung

Begleitmaterial zum Kurs

Stand: 20230414

Konventionen bei eigenen Projekten

BlueJ legt für jedes Projekt einen Ordner an. Dieser Ordnername ist der Projektname und kann selbstverständlich frei gewählt werden. Um ein Wirrwarr bei der Erstellung und Bewertung von Programmierprojekten zu vermeiden (Wer hat mitgearbeitet? Welche Version wurde abgegeben? Was wurde bearbeitet, was fehlt?), sollten folgende Vorgaben eingehalten werden:

● Jeder Projektname (auch, wenn man nur etwas für sich ausprobiert) beginnt immer mit einem umgekehrten Datum + Unterstrich. Beispiel: Für den 19. März 2012 wird **`20120319_`** verwendet. Dadurch kann man durch einfaches Sortieren eines Verzeichnisses eine zeitliche Reihenfolge herstellen. (Der Zeitstempel des Dateisystems ist dafür nur bedingt geeignet und geht oft beim zippen oder bei der Datenübertragung verloren).

● Bei einer Programmieraufgabe, die mehrere Teilaufgaben oder Teilschritte umfasst, sollte jede lauffähige Teillösung als Versions- bzw. Sicherheitskopie abgespeichert werden. Hintergrund: Arbeitet man ohne solche Kopien, können Programme nicht sinnvoll bewertet werden, denn am Stundenende (Gong) ist man oft mitten in der Arbeit und hat keine lauffähiges Programm vorzuweisen. Beispiel: Man beginnt mit der Aufgabe und erstellt z.B. das Projekt **`20120319_pacman`** . Hat man die erste Hürde erreicht (z.B. ein Spielfeld ist sichtbar, aber noch keine Figuren), so beendet man für einen Augenblick die BlueJ-Umgebung, kopiert den Projektordner und gibt der Kopie den Namen **`20120319_pacman_v1`** . Danach macht man mit dem ursprünglichen Projekt weiter. So entstehen immer Zwischenversionen, die die Endung v1, v2, v3 usw. tragen, so dass man erstens immer lauffähige Zwischenlösungen vorweisen kann und zweitens in einer Sackgassensituation auf frühere Versionen zurückgreifen kann.

● Sollen Projekte zur Bewertung abgegeben werden, so muss außerdem im Projektnamen noch der Name / die Namen der Programmierer auftauchen. Beispiel: **`20120319_pacman_tim_dennis`**
Dadurch können später alle Projekte in einem gemeinsamen Abgabeordner gespeichert werden.

● Je nach Vorgabe durch den Lehrer sind im Programmtext sinnvolle Kommentare zu setzen. Es ist nämlich sehr einfach, perfekt lauffähige, aber völlig unlesbare Programme zu schreiben. Zu den wichtigsten Quelltextkonventionen gehören:

* Lauf- und Zählvariablen heißen oft i,j,k, Koordinaten werden wie in Mathe oft mit x und y bezeichnet. Ansonsten „sprechende“ Namen benutzen!
* Variablen- und Methodennamen beginnen immer mit einem Kleinbuchstaben. Klassennamen beginnen immer mit einem Großbuchstaben. Durch „CamelCase“ können Namen zusammengesetzt werden. z.B.: **`meineTolleVariable`**, **`MeineTolleKlasse`**.

Hilfreiche Befehle zur Ein- und Ausgabe

Um aus einem Java-Programm etwas auszugeben (d.h. Text anzeigen zu lassen), benutzt man üblicherweise folgenden einfachen Befehl:

```
System.out.println("Hallo Welt");
```

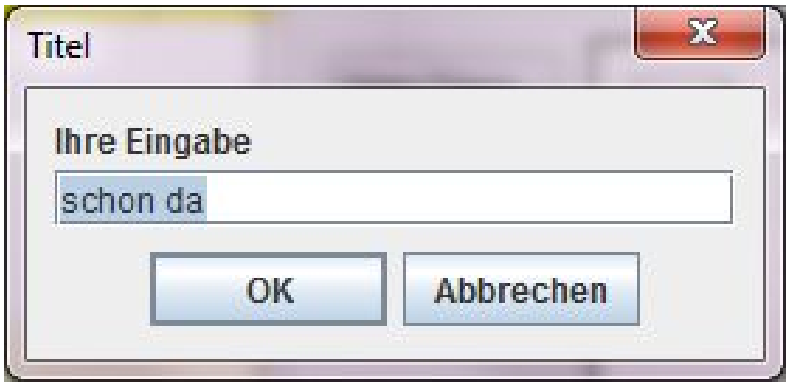
Oft will man nicht nur etwas ausgeben, sondern auch Benutzereingaben einlesen. Eine Möglichkeit ist die Verwendung von sogenannten Dialogfenstern. Dazu muss zu Beginn der Klassendatei (noch vor **class** ...) aber etwas importiert werden:

```
import javax.swing.JOptionPane;
```

Innerhalb eines Programms steht dann folgender Befehl zur Verfügung:

```
String s = (String) JoptionPane.showInputDialog
            (null,"Ihre Eingabe","Titel",
            JoptionPane.PLAIN_MESSAGE,
            null,null,"schon da");
```

Dieser Befehl erzeugt dann folgendes Fenster:



Die Parameter (auch Argumente genannt) der Methode **showInputDialog** dürften damit größtenteils erklärt sein. Wichtig ist, dass nach Bestätigung mit „OK“ oder Enter die Eingabe in der String-Variablen **s** zur Verfügung steht, mit der sich dann weiterarbeiten lässt.

Strings vergleichen: Möchte man Strings vergleichen, so kann man das nicht mit **„= =“** tun. Angenommen, man möchte prüfen, ob in einer String-Variablen **s** der Name unseres Landes enthalten ist, so muss man das wie folgt tun:

```
if(s.equals("Deutschland")) { ... }
```

Zahlen eingeben: Möchte man vom Benutzer eine Zahl erfragen, so haben wir das Problem, dass eine String-Variable keine Zahl ist, mit der wir direkt rechnen können. Um eine Stringvariable **s** in eine Zahl **i** umzuwandeln, schreibt man:

```
int i = Integer.parseInt(s);
```

Problematisch ist dabei, dass in **s** möglicherweise gar keine Zahl gespeichert ist, die umgewandelt werden könnte. In diesem Fall kommt es zu einem Programmabbruch (den wir erst später umgehen können).

Übungen:

1. Schreibe ein Programm, welches die Zahlen von 1 bis 100 ausgibt.

2. Schreibe ein Programm, welches den Benutzer nach einer Zahl fragt. Schließlich soll das Quadrat der eingegebenen Zahl ausgegeben werden.

Kommentare und Dokumentation mit JavaDoc

Arbeitet man an größeren Projekten, so weiß man nach zwei Wochen Arbeit oft nicht mehr, was man sich bei dem einen oder anderen Codeabschnitt gedacht hat. Das wird zum Problem, wenn man im Team arbeitet und jemand anders den eigenen Code lesen und verstehen muss. Wenn man also größere Projekte erfolgreich bearbeiten will, muss man gewissenhaft kommentieren und dokumentieren. Es gibt zwei Fälle zu unterscheiden:

1. Kommentierung von Programmabschnitten und kleineren Einheiten:

```
class Bruch {
    int z;       // Eigenschaft Zaehler
    /* Kommentar über
          mehrere Zeilen */
```

Beim Doppelschrägstrich gilt: Der Rest der Zeile ist Kommentar.
Bei Schrägstrich-Sternchen gilt: Solange Kommentar, bis **`*/`** auftaucht!
Bei solchen „kleinen Kommentaren“ gilt die Faustregel: Nicht das „Was“ sondern das „Warum“ soll geklärt werden.
Ein Kommentar der Art „Variable i wird mit dem Wert 8 belegt“ ist ziemlicher Unsinn, denn die Anweisung „**`i = 8;`**“ spricht für sich selbst. Viel informativer wäre z.B.: „i zählt die 8 Planeten unseres Sonnensystems durch“

2. Dokumentation von Klassen, Eigenschaften und Methoden mit JavaDoc

Die Java-API-Dokumentation, die im Netz zu finden ist, wurde automatisch generiert und steckt im Programmtext der Java-API! BlueJ hat einen JavaDoc-Dokumentationsgenerator eingebaut (Hauptfenster → Tools → Project Documentation), so dass wir die API-Dokumentation unserer eigenen Klassen direkt sehen können. Die wichtigsten „Tags“ (Schlüsselwörter) für JavaDoc sind:
`@version` und **`@author`** für Klassen,
`@param` und **`@return`** bei Methoden. JavaDoc-Kommentare fangen immer mit **`/**`** an und erstrecken sich ggf. über mehrere Zeilen:

```
/**
 * Bruch mit ganzzahligem Nenner und Zähler.
 * @author Bill Gates
 */
class Bruch {
    /**
     * Nenner und Zähler des Bruchs. n ist immer != 0
     */
    int n, z;
    /**
     * multipliziert Zähler & Nenner mit einem Faktor
     * @param faktor – damit wird multipliziert
     */
    void erweitereMit(int faktor) {  ......
```

Struktur eines einfachen Java-Programms

Java ist eine professionelle Programmiersprache. Mit dem Programm BlueJ (eine Entwicklungsumgebung für Java, also ein Programmierwerkzeug) lässt sich Java für Anfänger relativ gut erlernen. Ein (unvollständiges) Java-Programm könnte folgendermaßen aussehen:

```
01 import javax.swing.*;
02 /*
03  * .... (Kommentare) ....
04  */
05 public class Startklasse {
06     JFrame frame;
07         public void zeigeFenster() {
08          // Hier nun die eigentlichen Befehle...
09            frame = new JFrame();
10          frame.setTitle("Null Layout");
11            frame.setLayout(null);
12     }
13 }
```

Wir untersuchen dieses Java-Programm Zeile für Zeile:

Z01: Mit import können andere Programmteile (z.B. von fremden Programmierern) eingebunden und benutzt werden (Details später!)
Z02 bis Z04: ganze Blöcke können als Kommentar, d.h. als Information nur für den Programmierer dienen. Auf die Programmausführung haben sie keine Auswirkung. Auch **Z08** ist eine reine Kommentarzeile.
Z05: Java-Programme sind immer in sogenannten „Klassen“ organisiert. Hier wird bekannt gegeben, dass unsere eigene Klasse *Startklasse* heißen soll (sie könnte auch *SuperDuperGBG* heißen - der Name ist frei wählbar, solange er keine Leerzeichen/Sonderzeichen enthält). Beachte auch die öffnende geschweifte Klammer, die erst in **Z13** wieder geschlossen wird!

Z06: Die Variable frame (vom Typ JFrame) wird „angemeldet“.
Z07: Innerhalb von Klassen befinden sich u.a. Methoden, hier eine Methode namens *zeigeFenster*.
Z09 bis Z11: Die eigentlichen Befehle des Programms. Eure eigentliche Arbeit!
Z12: Schließende Klammer für die Methode zeigeFenster.

Möchte man ein Java-Programm ausführen, so muss man das Programm eigentlich erst abspeichern und compilieren. In BlueJ wird automatisch gespeichert, Compilieren muss man z.B. mit Rechtsklick auslösen.

Java ist bei Programmierfehlern sehr pingelig: Nicht geschlossene Klammern, fehlende Semikola, falsche Groß- oder Kleinschreibung führen zu Fehlern!

Variablen in Java

In Java (und anderen Programmiersprachen) sind Variablen Platzhalter für Werte (z.B. Zahlen, Buchstaben oder auch kompliziertere Daten wie Bilder o.ä.) die sich während des Programmablaufs ändern können. Variablen haben in Java immer einen Typ, der sich nicht ändert. Ganze Zahlen kann man in Variablen des **int**-Typs speichern:

```
1  int i, k;    // Typfestlegung für Variablen i und k
2  i = 4;       // i bekommt den Wert 4 zugewiesen
3  k = 5;       // k bekommt den Wert 5 zugewiesen
4  i = k + 7;   // i bekommt den Wert _____ zugewiesen
5  k = k+1;     // k wird ___________
6  2 = i;       // kompletter Unsinn! Fehlermeldung!
```

Bevor man Variablen benutzen kann, muss man deren Typ festlegen (Zeile 1). Das Gleichheitszeichen in den Zeilen 2 bis 5 ist nicht mathematisch zu verstehen, sondern als Zuweisungsbefehl.

Weitere wichtige Datentypen sind: **double** für „Kommazahlen“, **boolean** für die zwei Wahrheitswerte richtig/falsch (bzw. **true** oder **false**).

Anwendung: Wenn die Zufallszahl (0..9) kleiner als 2 ist, soll reagiert werden:

```
1  int zZahl;
2  zZahl = new Random().nextInt(10);
3  if(zZahl < 2) {
4    System.out.println("Glück! Zahl war"+zZahl);
5  }
```

In Zeile 3 ist die if-Bedingung **zZahl < 2** zu sehen, die eigentlich selbsterklärend ist. Desweiteren gibt es folgende Vergleichsoperatoren:

<= für „kleiner oder gleich“ (und **<** für „echt kleiner“)
>= für „größer oder gleich“ (und **>** für „echt größer“)
!= für „ungleich“
== für „genau gleich“ (Achtung: zwei Gleichheitszeichen!!)

Ein weiterer Variablentyp ist **String**. In solchen Variablen können Sätze (oder ganze Texte) gespeichert werden:

```
String ansage;
ansage = "anschnallen bitte!";
```

Man kann Strings mit + verketten (siehe Zeile 4 in obigem Programm). Möchte man testen, ob zwei Strings gleich sind, so muss man (statt ==) **equals** nutzen:

```
wort = "Pech";
if(wort.equals("Glück")) {
  // Block wird nicht ausgeführt, da "Pech" ≠ "Glück"
}
```

Atomare Datentypen

Datentyp `int` (und auch `long`): Variablen diesen Typs speichern (positive und negative) ganze Zahlen. Oft benutzt man sie, um Dinge zu zählen - insbesondere bei einer Zählschleife. Man kann mit ihnen rechnen: **+** (plus) **-** (minus) und ***** (mal) funktionieren wie gewohnt. Die Operation **/** (geteilt) ist aber eine Ganzzahl-Division, die „wie in der Grundschulmathematik" funktioniert: Dort hat man wie folgt gerechnet: 27 geteilt durch 4 ist gleich 6 Rest 3 (denn 6*4+3 =27). In Java sieht das folgendermaßen aus:

```
int a = 27 / 4;          // a bekommt den Wert 6 zugewiesen
```

Datentyp `boolean`: Variablen diesen Typs können nur die Werte **`true`** oder **`false`** speichern. Man benutzt sie oft um Bedingungen zu formulieren weil man sie mit den Operatoren **&&** (und), **||** (oder), **!** (nicht) verknüpfen kann:

```
boolean hungrig = true;          boolean durstig = false;
if (hungrig && durstig) { ... }
```

Datentyp `double` (und auch `float`): Variablen diesen Typs speichern reelle Zahlen - aber nur annähernd! Man benutzt sie für allgemeine Rechnungen bei denen auch gebrochene Zahlen vorkommen. Die Operationen **+** (plus) **-** (minus), ***** (mal) und **/** (geteilt) funktionieren wie gewohnt. Bei „Kommazahlen" wird statt des Kommas ein Punkt verwendet:

```
double ungefaehrWurzel2 = 1.41;
double ungefaehrPi = 3.14159;
```

Achtung: Ein Computer (ebenso wie ein Taschenrechner) macht in Spezial-fällen bei Kommazahlen (vom Typ `double`) technisch bedingte Rechenfehler. Als Faustregel gilt: Wenn `int`-Zahlen ausreichen, sollte man keine `double`-Zahlen verwenden. Insbesondere bei Zählschleifen!

Umwandlung von atomaten Datentypen (Typecasting)

Möchte man int- und double-Zahlen in einer Rechnung vermischen, so muss man möglicherweise Typumwandlungen vornehmen:

```
double a = 1.2;
int i = 2 + a;          // Fehlermeldung!
```

Da in der zweiten Zeile das Ergebnis eigentlich die „Kommazahl" 3.2 ist, welche aber nicht als int-Zahl abgespeichert werden kann, erzeugt Java eine Fehlermeldung, damit man dazu gezwungen wird, sich über das Problem Gedanken zu machen. Oft ist die Lösung, den Wert einfach in den passenden Datentyp umzuwandeln und dabei möglicherweise ungenau zu werden:

```
int j = (int)(2 + a); // Funktioniert! j ist jetzt 3
```

Oft tritt das Problem bei Benutzung der Java-API auf, wenn bestimmte Typen verlangt werden, z.B.: **`public static long currentTimeMillis()`**

Mit Zahlen und Zeiten arbeiten

Bekannt aus der Grundschule ist die **Division mit Rest**:

19 ÷ 5 = 3 Rest 4 | 3 ÷ 5 = 0 Rest 3 | 14 ÷ 7 = 2 Rest 0

In Java wird die **Division bei int-Zahlen** folgendermaßen durchgeführt:

`int i = 19/5;`	`int j = 3/5;`	`int k = 14/7;`
i erhält den Wert 3	j erhält den Wert 0	k erhält den Wert 2

Mit dem **Modulo-Operator % berechnet man den Divisions-Rest**:

`int i = 19%5;`	`int j = 3%5;`	`int k = 14%7;`
i erhält den Wert 4	j erhält den Wert 3	k erhält den Wert 0

Oft muss man in Java überprüfen, ob eine bestimmte Zeitspanne verstrichen ist:

```
long zeitA = System.currentTimeMillis();
```

In **`zeitA`** ist nun die Zahl der seit dem 1.1.1970 verstrichenen Millisekunden gespeichert. Liest man diese Systemzeit zu einem späteren Zeitpunkt nochmal aus, so kann man die Zeitdifferenz zwischen den Aufrufen bestimmen:

```
long zeitB = System.currentTimeMillis(); //später
long zeitDiff = zeitB-zeitA; // Differenz in ms
int zeitDiffZehntel = (int) (zeitDiff / 100);
```

Nun kann es sein, dass man jede Zehntelsekunde eine bestimmte Animations-phase darstellen möchte. Z.B.: Falls der „Zeiger“ auf 0 Zehntelsekunden steht → SmileyAuge offen, Zeiger auf 1 Zehntelsekunde → SmileyAuge schließt sich, usw.. Und das ganze soll sich wiederholen, wenn eine Sekunde um ist.
Man benötigt also Zahlen zwischen 0 und 9:

```
int zehntelSekundenPhase =  zeitDiffZehntel%10;
```

Obiges Verfahren scheint vielleicht etwas kompliziert. Wenn man genau weiß, dass man einfach nur warten muss, und währenddessen nichts anderes passieren soll (Achtung: für GUI-Anwendungen daher oft ungeeignet), benutzt man:

```
try{ Thread.sleep(3000); } catch(Exception e) {}
// für 3000ms (3 Sekunden) warten und nichts tun.
```

Bei der Arbeit mit z.B. **`currentTimeMillis()`** erhält man Werte, die oft (wie oben) angepasst und umgerechnet werden müssen. Häufig muss man auch erzeugte Zufallszahlen umrechnen. Die Methode **`Math.random()`** erzeugt „Kommazahlen“ (also vom Typ **`double`**) zwischen 0.0 und 1.0. Um daraus eine **`int`**-Zahl im Bereich von -10 bis 10 zu erzeugen, geht man so vor:

```
double z = Math.random();     // z zwischen 0.0 und 1.0
int k = (int) (z * 21);       // k zwischen 0 und 20
int h = k-10;                 // h zwischen -10 und 10
```

Methoden in Java-Programmen: Teil 1

In Java kann man öfters genutzte Programmabschnitte in eine sogenannte **Methode** auslagern. Diese kann man je nach Bedarf aufrufen. Schauen wir uns noch einmal die Grobstruktur eines Java-Programms an:

```
01 public class MeineKlasse {
02        public static void starten() {
03         ..... // Eure Befehle
04     }
05 }
```

Angenommen, wir möchten öfters ein (immer gleiches) Teilproblem lösen. Bislang mussten wir dann an jeder solchen Stelle den betreffenden Programmteil einfügen. Dadurch werden Programme schnell unübersichtlich. Besser ist es, auf Wiederholungen zu verzichten und Methoden zu benutzen:

```
public static void zeigeFehlermeldung() {
  System.out.println("********* Achtung *********");
  System.out.println("Nur ja/nein-Antwort möglich");
}
```

Dies allein bewirkt noch gar nichts. Jedoch können wir nun im eigentlichen Programm (Zeile 3 etc.) notieren: `zeigeFehlermeldung();`
und dies beliebig viele Male an verschiedenen Stellen.

Methoden mit Argumenten

Oft möchte man Methoden vielseitiger gestalten. Zwar könnten wir eine Methode `fünfLeerzeilenAusgeben()` schreiben, jedoch wäre es in diesem Fall vielseitiger, wenn man die Anzahl der Leerzeilen erst beim Methodenaufruf angibt, in etwa so: `leerzeilenAusgeben(5);`
Dies ist möglich, wenn man die Methode wie folgt schreibt:

```
public static void leerzeilenAusgeben(int anzahl) {
  for(int i = 0; i<anzahl; i++) {
    System.out.println();
  }
}
```

Der bei Methodenaufruf angegebene Wert ist in der Methode nun in der Variable `anzahl` benutzbar (welche den Typ `int` hat). Man nennt die „übergebene“ Variable auch **Argument** (der Methode) oder Parameter.

Der String `"********* Achtung *********"` im Beispiel weiter oben ist übrigens ein Argument der Methode println!

Methoden in Java-Programmen: Teil 2

Wir haben gelernt, wie man eigene Methoden in Java schreibt. Wir haben außerdem gelernt, dass man den Methoden Variablen „mitgeben“ kann (sogenannte Argumente oder Parameter). Ein Beispiel dafür war die Methode

```
public static void leerzeilenAusgeben(int anzahl)
```

die dafür sorgt, dass genau soviele Leerzeilen ausgegeben werden, wie im Parameter **anzahl** angegeben wurde (Siehe voriges Blatt).

Methoden können aber nicht nur Parameter entgegennehmen, sondern auch selbst nach Ausführung ein „Ergebnis“ (d.h. einen Rückgabewert) zurückgeben!

Bevor wir eigene Methoden mit Rückgabewerten schreiben, wollen wir derartige bestehende Methoden nutzen. Java stellt z.B. die Methode **random** aus der Klasse **Math** zur Verfügung, welche eine Zufallszahl zwischen 0 und 1 liefert. Dies könnte man folgendermaßen nutzen:

```
double z1 = Math.random();
System.out.println("Zufallszahl ist: "+ z1);
```

Hinweis: Wenn wir Methoden aus fremden Klassen nutzen, müssen wir auch den Klassennamen (bzw. den Objektnamen) mit angeben. Ruft man Methoden aus der „eigenen“ Klasse auf, so kann man den Klassennamen weglassen.

Methoden mit Rückgabewert können wir ganz einfach selbst schreiben, indem wir folgendermaßen vorgehen:

```
01 public static int zufallZwischen(int a, int b) {
02     double z = Math.random();
03     int diff = b - a;
04     int z2 = (int) Math.floor(z * diff);
05     int ergebnis = a + z2;
06     return ergebnis;
07 }
```

Erklärung: Die Methode liefert uns eine Zufallszahl zwischen a und b.
In Z01 ist nun zu Beginn statt **public static void** die Formulierung **public static int** zu sehen. Damit gibt man bekannt, dass die Methode einen **int**-Rückgabewert hat (statt - bei **void** - gar keinen). In Z02 bis Z05 passiert etwas Mathematik und am Ende der Methode wird in Z06 das Ergebnis mit der return-Anweisung zurückgegeben.

Wichtig: Der Typ des Rückgabewertes kann auch **boolean** (true oder false) sein. Ein Wert diesen Typs kann als Ergebnis direkt in einer if-Abfrage verwendet werden!

Aufgabe: Schreibe folgende Methode und benutze sie:

```
public static boolean istQuadratzahl(int zahl)
```

Grundlegende Kontrollstrukturen in Java

Um den Programmfluss in Java zu steuern, stehen ein paar grundlegende Befehle, die sogenannten Kontrollstrukturen zur Verfügung. Die Wichtigsten:

1. Verzweigung:

```
if (i == 5) {
        System.out.println("Variable i ist 5");
} else {
        System.out.println("Variable i ist nicht 5");
}
```

In runden Klammern hinter der **if**-Anweisung muss eine Bedingung stehen, die entweder wahr oder falsch ist. Den **else**-Zweig kann man auch weglassen.

2a. Zählschleife:

```
for (int i=2; i<100; i++) {
        System.out.println("Zahl: "+ i);
}
```

Die for-Schleife wird in obigem Beispiel 98 mal durchlaufen. Dabei nimmt i die Werte von ___ bis ____ an. Die Struktur in der **`for`**-Zeile bedeutet:

→ Initialisierung der Zählvariable **i** vom Typ **integer** auf den Anfangswert 2.
→ Schleifenbedingung (**i<100**): Falls erfüllt, wird der Rumpf ausgeführt.
→ **i++** (d.h. **i** um 1 erhöhen): Wird nach jedem Schleifendurchlauf ausgeführt.

2b. While-Schleife

```
double x = 0;
while (x <= 0.9) {
        x = Math.random(); // Zufallszahl zw. 0 und 1
}
```

Die while-Schleife wird solange ausgeführt, wie die Zufallszahl kleiner als 0.9 ist. Wie bei der **if**-Anweisung steht hinter dem **while** eine Bedingung.

Hinweise zu Bedingungen:

Die erwähnten Bedingungen sind Ausdrücke, die wahr oder falsch sein können, also den boolean-Wert **`true`** oder **`false`** haben. Daher kann man überall dort, wo Bedingungen auftauchen, auch z.B. das Ergebnis einer Methode verwenden, die einen Boolean-Wert zurückgibt:

```
if (meinString.equals("Hallo")) { ..
while(meinString.equals("Hallo")) {..
for(int i=0; meinString.equals("Hallo"); i++) {..
```

Alle o.g. Anweisungen sind gültig, wobei die for-Schleife in obigem Beispiel unüblich und damit schlecht lesbar bzw. schlechter Stil wäre.

Bedingungen kann man kombinieren mit folgenden booleschen Operatoren:
„ist gleich“ **==**, „ist ungleich“ **!=**, „ist kleiner als“ **<**, „ist kleiner gleich“ **<=**, „nicht“ **!**, „oder“ **||**, „und“ **&&**

Klassen und Objekte

In der deutschen Umgangssprache unterscheiden wir oft zwischen einer Kategorie (oder einem Oberbegriff) wie z.B. „Säugetier“ und einem konkreten Vertreter dieser Kategorie, z.B. „ein Löwe“.
Die Idee, solche Kategorien/Oberbegriffe als „Schablone“ für einzelne tatsächliche Dinge zu benutzen, hat sich in der Informatik als äußerst hilfreich erwiesen: In der Informatik bzw. in Java benutzt man dafür die Wörter **Klasse** (die Schablone oder Kategorie oder der Oberbegriff) und **Objekt** (das tatsächliche Ding, ein Vertreter einer Kategorie).

Beispiele: Der Bär Balu (Objekt) aus dem Dschungelbuch ist ein Säugetier (Klasse).
Der Hund Lassie (aus der Fernsehserie) ist ein Säugetier (Klasse).
„Der Herr der Ringe“ (Objekt) ist ein Buch (Klasse). Die Bibel ist ein Buch etc.

Übung: Finde drei „Klassen“ mit jeweils drei unterschiedlichen „Objekten“!

Wenn wir in der Umgangssprache von einem „Säugetier“ sprechen, so haben wir instinktiv schon gewisse Vorstellungen von den Eigenschaften eines Säugetiers - selbst wenn wir nicht über einen konkreten Löwen oder Elefanten reden. Ein Säugetier hat z.B. die Eigenschaft „Geschlecht“ und ist entweder männlich oder weiblich. Bei einem Buch ist die Eigenschaft Geschlecht unsinnig. Ebenso können wir bzgl. „Säugetieren“ bestimmte Fragen („alterInJahren“) stellen oder bestimmte Handlungen ausführen („fütternMitXKiloFleisch“).
In Java sprechen wir ebenso von **Eigenschaften**, und die Fragen oder Handlungen sind in Java die **Methoden**.

Beispiel: „Der Medicus“ (Objekt der Klasse Buch) hat die Eigenschaften „Seitenzahl“, „Alter“, „Verlag“. Und Methoden könnten sein: „blättereUm“, „reißeSeiteNrXheraus“, „übersetzeSeiteNrXinsRussische“

Übung: Finde zu jeder Deiner Klassen jeweils drei Eigenschaften und drei Methoden.

In Java gibt es eine Klasse **`JButton`** (Bitte mit **`import javax.swing.*;`** einbinden). Ein Objekt der Klasse **`JButton`** hat viele Eigenschaften (z.B. den enthaltenen Text) und u.a. folgende Methoden:

```
void   setText(String s)
String getText()
void   setBounds(int x, int y, int width, int height)
```

Anders als einfache Variablen wie z.B. int muss man bei Objekten nicht nur den Typ deklarieren, sondern auch ein Objekt anlegen:

```
JButton a;
a = new JButton();
```

Arrays in Java

Ein Array (auch Feld genannt) ist ein spezieller Datentyp, der mehrere Werte (gleichen Typs) zu einer Einheit zusammenfasst. Angesprochen werden die Elemente über einen ganzzahligen Index (also deren Position, Zählung beginnt mit der Null). Jeder Platz nimmt immer einen Wert des gleichen Typs auf (in einem int-Array können keine boolean-Werte gespeichert werden).

Jedes Array beinhaltet Werte nur eines bestimmten Datentyps bzw. Grundtyps. Dies können sein:

- elementare Datentypen wie int, byte, long und so weiter
- Referenztypen (d.h. Objekte)
- Andere Arrays, um „Arrays von Arrays“ zu realisieren.

Eine Array-Variablendeklaration ähnelt einer gewöhnlichen Deklaration, nur dass nach dem Datentyp eckige Klammern [und] gesetzt werden:

```
int[]   primzahlen;
Point[] meinePunkte;
```

Die Variable **primzahlen** hat jetzt den Typ »ist Feld« und »speichert int-Elemente«, also eigentlich zwei Typen.

Array-Objekte erzeugen

Ein Array muss mit dem new-Operator unter Angabe einer festen Größe erzeugt werden. Die Deklaration der Variablen allein erzeugt noch kein Feld mit einer bestimmten Länge. Die Länge des Felds wird in eckigen Klammern angegeben. Hier kann ein beliebiger Integer-Wert stehen, auch eine Variable. Selbst 0 ist möglich. Beispiel:

```
int[] values;
values = new int[ 10 ];
```

Die Feld-Deklaration ist auch zusammen mit der Initialisierung möglich:

```
double[] values = new double[ 10 ];
```

Ebenso kann man kurze Felder direkt belegen:

```
char[] name  = { 'T', 'i', 'm' };
```

Achtung: Die Operatoren == und != haben ihre Objekt-Bedeutung: Sie vergleichen, ob zwei Variablen auf das gleiche Array-Objekt verweisen, aber nicht die Inhalte der Arrays (das kann aber **equals()** aus der Klasse **Arrays**).

Zugriff auf die Array-Elemente

```
char[] name  = { 'C', 'h', 'r', 'i', 's' };
char   first = name[ 0 ];                    // C
char   last  = name[ name.length - 1 ];      // s
for(int i = 0; i<name.length; i++) {
      name[i] = 'a';   // alles mit 'a' belegen
}
```

Array-Objekte haben die Eigenschaft `length`, um deren Länge zu ermitteln.

Arrays in Java Teil 2: Arrays von Objekten

Ein Array kann nicht nur Daten mit elementarem Typ (wie int, byte, long und so weiter) aufnehmen, sondern auch Objekte, wie beispielsweise Strings.

Beispiel 1: Stringarrays in der Java-API

Gelegentlich hat man das Problem, dass man in Java Zeichenketten auswerten muss, in denen durch Komma getrennte Werte stehen (sogenannte CSV-Dateien, also CommaSeparatedValues). Beispielsweise könnte man eine Liste bekommen, in der erst ein Schülernachname, dann dessen Vorname und schließlich dessen Alter aufgeführt ist:

```
Mustermann, Max, 17
Becker, Birgit, 15
…usw.
```

Möchte man nun eine einzelne Zeile im Programm untersuchen, so muss man irgendwie an die einzelnen Werte herankommen, die ja durch Komma getrennt sind. Weil dies ein Standardproblem ist, gibt es dafür schon eine Lösung in der Klasse String: Die Methode „split“ kann unsere Zeile an den Kommazeichen aufspalten und die einzelnen Werte werden als String-Array zurückgegeben:

```
String zeile  = "Mustermann, Max, 17";
String[] werteArray = zeile.split(",");
String nachname = werteArray[0];
String vorname = werteArray[1];
String alter = werteArray[2];
```

Diese einzelnen Werte kann man dann weiterverarbeiten.

Beispiel 2: Stringarray in der main-Methode

Die main-Methode eines Java-Programms hat immer folgende Gestalt:

```
public static void main(String[] args) {...
```

Es steht also bei Programmstart ein String-Array zur Verfügung. Was steht da drin? Antwort: Bei allen ausführbaren Programmen (nicht nur Java!) kann man beim Programmstart sogenannte Kommandozeilenparameter angeben (bei einigen Computerspielen kann man damit vor Spielstart z.B. die Bildschirmauflösung einstellen). Der Aufruf über die Kommandozeile sieht dann möglicherweise so aus:

```
pacman.exe -res 320x200
```

Auch Java-Programme können ohne BlueJ gestartet werden. Eine Möglichkeit ist es, auf dem Desktop eine Verknüpfung folgender Form anzulegen:

```
java.exe -cp "D:\pfad\zur\klasse" Pacman -res 320x200
```

Ist nun Pacman eine Java-Klasse (Pacman.class), die im angegebenen Verzeichnis existiert, so wird diese ausgeführt. Die Parameter „-res“ und „320x200“ sind nun in der Java-main-Methode die Elemente des o.g. **`args`**-Arrays.

Beispiel 3: Ein Spielfeld als Stringarray

Ein Spielfeld mit quadratischen Feldern (z.B. Schach) könnte man als Array speichern, in dem 8 Strings mit jeweils 8 Zeichen enthalten sind. Die einzelnen Zeichen des Strings stehen dann für Spielfiguren ('K'=King, 'Q'=Queen usw.)

Eigene Klassen, eigene Methoden – Teil 1: Schreiben

Angenommen, man möchte in Java mit Brüchen rechnen können. Java stellt eine solche Möglichkeit nicht direkt bereit. Aber wir können uns eine Klasse namens **`Bruch`** schreiben, mit der sich Brüche (und das Rechnen damit) darstellen lassen:

```
01 class Bruch {
02     int z;       // Eigenschaft Zaehler
03     int n;       // Eigenschaft Nenner
04
05     void erweitereMit(int faktor) {
06         z = z * faktor;
07         n = n * faktor;
08     }
09
10     int getZaehler() {
11         return z;
12     }
13
14     int getNenner() {
15         return n;
16     }
17
18     Bruch(int zaehler, int nenner) {
19         z = zaehler;
20         n = nenner;
21     }
22 }
```

Diese Klasse ist ein absolutes Minimalbeispiel (und noch wenig brauchbar); Sie enthält aber alle wichtigen Bestandteile einer Klasse:

1. In Zeile 1 wird der Java-Umgebung bekanntgegeben, dass nun der Quelltext für die Klasse **`Bruch`** folgt. Die geschweifte, öffnende Klammer wird in Zeile 22 wieder geschlossen. Alles dazwischen gehört zur Klasse.
2. In Zeile 2 und 3 werden Objektvariablen (**Eigenschaften**) deklariert. Diese Variablen sind in allen Teilen der Bruch-Klasse bekannt und veränderbar.
3. In Zeile 5, 10 und 14 werden **Methoden** definiert. Eine Methode funktioniert wie eine BlackBox: Man „steckt etwas hinein“ (die sog. **Argumente**. Die Integervariable **`faktor`** ist ein Argument der Methode **`erweitereMit`**); In der Methode passiert irgendetwas und man bekommt einen **Rückgabewert** zurück (z.B. einen Integerwert für den Nenner in Zeile 14. DieMethode muss deswegen eine **`return`**-Anweisung enthalten). Erwartet eine Methode keine Argumente, bleibt die Argumentliste leer (Zeile 10 und 14). Gibt eine Methode nichts zurück, so wird **`void`** benutzt (Zeile 5)
4. Der **Konstruktor** in Zeile 18 wird beim Anlegen des Objekts aufgerufen.

Eigene Klassen, eigene Methoden – Teil 2: Benutzen

Da wir nun eine Bruch-Klasse zur Verfügung haben, möchten wir sie auch benutzen. Wir betrachten folgende Testklasse:

```
01 class TestBruch {
02
03   public static void main(String[] args) {
04     Bruch dreiviertel;
05     Bruch zweiachtel;
06
07     dreiviertel = new Bruch(3,4);
08     zweiachtel = new Bruch(2,8);
09
10     int multZaehler = dreiviertel.getZaehler() *
                              zweiachtel.getZaehler();
11     int multNenner = dreiviertel.n * zweiachtel.n;
12 } }
```

Wir gehen dieses Testprogramm Zeile für Zeile durch:

1. Auch das einfachste Testprogramm kann nicht allein für sich stehen, sondern muss Teil einer Klasse sein. Daher Zeile 1.
2. Die genaue Bedeutung aller Teile von Zeile 3 kann an dieser Stelle nicht erklärt werden. Aber jedes Java-Programm braucht einen „Anfang“, einen Einstiegspunkt. Dieser ist die **`main`**-Methode, mit der die Ausführung eines jeden Java-Programms begonnen wird.
3. In Zeile 4 und 5 werden zwei Variablen mit Namen **`dreiviertel`** und **`zweiachtel`** **deklariert** (nicht definiert!), d.h. ihr Typ wird festgelegt.
4. In Zeile 7 und 8 werden die Brüche schließlich **definiert**. Genauer: Es werden zwei **Objekte** vom Typ **`Bruch`** angelegt, indem der entsprechende Konstruktor aufgerufen wird.
5. In Zeile 10 werden die Zähler der Brüche multipliziert. Dazu wird von jedem Bruch die **`getZaehler()`**-Methode benutzt: Diese erwartet keine Argu-mente, hat aber als Rückgabewert eine Integerzahl (siehe Klasse **`Bruch`**).
   Ebenfalls möglich ist es, direkt auf die Eigenschaften der **`Bruch`**-Klasse zuzugreifen, wie es in Zeile 11 geschieht: Anstatt die Methode **`getNenner()`** aufzurufen (was ebenfalls möglich gewesen wäre) wird hinter dem Punkt einfach die Objektvariable geschrieben.

Bemerkung: Konstruktoren heißen immer genauso wie ihre Klassen. Klassennamen beginnen immer mit einem Großbuchstaben, Methodennamen jedoch mit einem Kleinbuchstaben. Man kann in Zeile 7 und 8 also an drei Dingen erkennen, dass es sich um eine Objekterzeugung handelt, und nicht um einen einfachen Methodenaufruf: Am **`new`**-Operator, am „Methodennamen“ **`Bruch`** (der genauso heißt wie die Klasse) und an der Tatsache, dass **`Bruch`** – anders als andere Methodennamen – mit einem Großbuchstaben beginnt.

Konstruktoren, `new` und `null`

Sobald man in Java nicht mehr mit den primitiven Datentypen (**`int, double, boolean`** etc.) auskommt, weil man mit komplizierteren Dingen hantieren möchte (z.B. hat bereits ein Bruch Nenner und Zähler, die eine Einheit bilden), so benötigt man Objekte, die – anders als die o.g. primitiven Typen – erst mit **`new`** angelegt werden müssen:

```
Bruch dreiviertel;                // Zeile 1
dreiviertel = new Bruch(3,4);     // Zeile 2
dreiviertel.erweiternMit(5);      // Zeile 3
Bruch b = dreiviertel;            // Zeile 4
```

Wir nehmen an dieser Stelle einmal an, dass jemand die Klasse **`Bruch`** bereits geschrieben hat. (Wie so etwas geht, behandeln wir später.)
In der ersten Zeile des obigen Programmausschnitts wird die Variable mit dem (vom Programmierer frei wählbaren) Namen **`dreiviertel`** vom Typ **`Bruch`** deklariert, d.h. Die Java-Umgebung weiß nun, welchen Typ die Variable hat. Weil **`Bruch`** kein primitiver Datentyp (sondern ein Referenztyp) ist, kann man der Variablen **`dreiviertel`** nicht einfach einen Wert zuweisen, wie man das bei einfachen Integerzahlen tun konnte (z.B.: **`int i = 28;`**). Stattdessen muss man ein **`Bruch`**-Objekt erst mit dem **`new`**-Operator anlegen (Java sorgt dann für ausreichende Reservierung von Speicher für das Objekt).

Dass man beim Anlegen unseres Bruches direkt die Werte für Zähler und Nenner mitgeben konnte, liegt an der Programmierung der Bruch-Klasse. Dort gibt es nämlich einen **Konstruktor** (eine „spezielle Methode“), welcher zwei Werte (für Zähler und Nenner) akzeptiert. Allgemein gilt: Immer wenn ein Objekt mit **`new`** angelegt wird, wird ein Konstruktor aufgerufen. Es ist möglich, dass eine Klasse unterschiedliche Konstruktoren hat (die aber immer genauso heißen wie die Klasse und sich nur in ihrer Argumentliste unterscheiden). Z.B. wäre vorstellbar, dass die Bruch-Klasse auch einen Konstruktor für gemischte Zahlen bereitstellt: **`Bruch eindreiviertel = new Bruch(1,3,4)`**

In Programmzeile 3 wird die Methode **`erweiternMit()`** auf unser Objekt aufgerufen. Dies klappt nur, weil in Zeile 2 tatsächlich ein Objekt angelegt wurde. Hätte man Zeile 2 einfach ausgelassen, so würde Java mit einer Fehlermeldung abbrechen, weil man auf nicht-existierende Objekte keine Methoden anwenden kann. Die Fehlermeldung „null-Pointer Exception“ deutet an, dass eine deklarierte, aber nicht definierte, Variable eines Referenztyps zunächst den Standardwert **`null`** erhält. Dahinter verbirgt sich sozusagen ein „Nicht-Objekt“ (was durchaus sehr sinnvoll eingesetzt werden kann, z.B. als Zeichen dafür, dass irgendetwas nicht geklappt hat (z.B. „Datei nicht vorhanden“)).

Zu guter Letzt: In Zeile 4 wird die Variable **`b`** sowohl deklariert als auch definiert. Wichtig ist, dass nach wie vor nur ein Objekt im Speicher existiert, welches aber von zwei Variablen referenziert wird. (Dazu später mehr!)

Die spezielle Referenz `this`

Wir nehmen als Beispiel eine Klasse **`Spielfigur`**, deren Objekte ihre eigenes Alter kennen:

```
public class Spielfigur {
	int alter;
	public void setAlter(int a) {
		alter = a;
	}
	// hier fehlen noch weitere Methoden...
}
```

Außerdem legen wir irgendwo zwei Objekte der Klasse an:

```
	Spielfigur donald = new Spielfigur();
	donald.setAlter(45);
	Spielfigur dagobert = new Spielfigur();
	dagobert.setAlter(65);
```

Nun möchte man eine Methode schreiben, die den Altersunterschied der beiden Figuren ermittelt. Sinnvoll wäre es, eine entsprechende Methode in die Klasse Spielfigur einzufügen – schließlich geht es um den Altersunterschied von Spielfigur-Objekten. Und elegant wäre es, wenn man den Altersunterschied folgendermaßen ermitteln könnte:

```
	unterschied = donald.altersdiff(dagobert);   (*)
```

Die erste Zeile (man nennt sie *„Signatur“*) der Methode, die wir schreiben wollen, müsste demnach folgendermaßen aussehen:

```
	public double altersdiff(Spielfigur s)
```

Ein Problem taucht nun beim Schreiben der Methode **`altersdiff`** auf, denn wir müssen auf das Alter beider Objekte zugreifen. Man kann zwar innerhalb der Methode mit **`s.alter`** auf das Alter des übergebenen Objektes zugreifen (im *(*)*-Beispiel also auf die Koordinaten von **`dagobert`**, denn **`s`** ist innerhalb der Methode eine Referenz auf **`dagobert`**). Aber wir brauchen innerhalb der Methode auch eine Referenz auf dasjenige Objekt, auf welches die Methode aufgerufen wurde (im *(*)*-Beispiel also auf das **`donald`**-Objekt). Und für dies Objekt gibt es innerhalb der Methode die spezielle this-Referenz! Die komplette Methode sieht nun so aus:

```
	public double altersdiff(Spielfigur s) {
		return Math.abs(s.alter - this.alter);
	}
```

Mit anderen Worten: Man benutzt **`this`** innerhalb einer Methode X, wenn man eine Referenz auf dasjenige Objekt benötigt, bei dem die Methode X aufgerufen wurde. Der eigentliche Objektname ist höchstens außerhalb der Klasse unter einem Variablennamen bekannt.

„Sichtbarkeit“ und Gültigkeitsbereiche von Variablen

Betrachte noch einmal die Klasse **`Bruch`** und **`TestBruch`** aus den beiden Kapiteln zu „eigene Klassen, eigene Methoden“. Bei der Klasse **`Bruch`** gibt es die Eigenschaften **`z`** und **`n`** (also Zähler und Nenner), die zwar innerhalb der Klasse, aber *außerhalb* jeder Methode deklariert werden.
In Klasse **`TestBruch`** werden jedoch die Variablen *innerhalb* einer Methode deklariert bzw. definiert (in Zeile 4, 5, 10 und 11). Wo liegt der Unterschied?

Objektvariablen / Eigenschaften (wie **`z`** und **`n`** in **`Bruch`**) sind in jeder Methode der Klasse direkt benutzbar und veränderbar. Sie „vergessen“ ihre Werte nicht, solange das Objekt existiert, in welchem sie deklariert worden sind. (Allerdings haben Eigenschaften verschiedener Objekte in der Regel auch verschiedene Werte: Z.B. die Eigenschaftswerte der **`Bruch`**-Objekte **`dreiviertel`** und **`zweiachtel`** aus der Klasse **`TestBruch`**)

Die Variablen **`dreiviertel`** oder **`multZaehler`** aus **`TestBruch`** sind nur in der **`main`**-Methode gültig. Ist die Methode beendet, sind auch die Werte der Variablen verloren (bzw. werden die innerhalb der Methode erzeugten Objekte wieder „vergessen“ und zerstört). Man nennt sie **lokale Variablen**. Dieses Verhalten zeigt sich sogar für beliebige Blöcke aus geschweiften Klammern:

```
01 if (Math.random() < 0.5) {
02   int i = 4;
03   System.out.println("i ist: "+ i);
04 }
05 System.out.println("i ist: "+ i);  // Fehler!
```

Hier wird innerhalb des if-Blocks eine int-Variable deklariert die auch nur dort gültig ist. Weil die Variable außerhalb des if-Blocks (Zeile 5) unbekannt ist, compiliert Java den obigen Programmabschnitt nicht!

Es ist sehr nützlich, dass Objektvariablen „methodenübergreifend“ bekannt sind, während oft benutzte Variablen wie **`i`** sinnvollerweise als lokale Variablen „mehrfach“ benutzt werden können.

Der **Modifizierer `private`**: In der Klasse **`TestBruch`** wird in Zeile 11 die Objekteigenschaft **`n`** direkt benutzt. Dies ist meistens schlechter Stil:
Die Eigenschaft **`n`** kann nämlich nicht nur gelesen, sondern auch direkt verändert werden. Würde ein dummer Programmierer nun die Zahl Null als Bruch erzeugen wollen, so könnte er auf die Idee kommen, der Eigenschaft **`n`** den Wert 0 zuzuweisen. Ein Bruch mit Nenner =0 ist jedoch nicht erlaubt! Damit solche gefährlichen Ausnahmesituationen erst gar nicht auftauchen, deklariert man die Variablen **`z`** und **`n`** der Klasse **`Bruch`** als **`private`**, also als „nur sichtbar innerhalb der Klasse“: **`private int n,z;`**
Damit wäre die Zeile 11 aus **TestBruch** nicht mehr erlaubt. Stattdessen müsste man in der Klasse Bruch die Methode **setNenner(int n)** einfügen, welche auf den unerlaubten Wert 0 angemessen reagiert.

Methoden überladen

In der Klasse Bruch könnte folgende Methode existieren:

```
void multipliziereMit(int zaehler, int nenner) {
    z = z * zaehler;
    n = n * nenner;
}
```

Hat man in einem Hauptprogramm nun z.B. ein Bruchobjekt **b** angelegt, so lässt sich dieser Bruch mit dem **zaehler** und **nenner** eines weiteren Bruchs multiplizieren (wobei Bruchobjekt **b** schließlich das Ergebnis enthält). Die Sache sähe etwa so aus:

```
Bruch b = new Bruch(1,2);
b.multipliziereMit(3,4);   // zwei Argumente: 3 und 4
```

Hat man aber bereits einen Bruch **b2** (welcher z.B. den Wert dreiviertel schon enthält), so müsste man für eine Multiplikation einen Umweg nehmen:

```
b.multipliziereMit(b2.getZaehler(),b2.getNenner());
```

Dies ist offensichtlich unschön. Eleganter wäre folgende Lösung:

```
b.multipliziereMit(b2);    // elegant!
```

Nun hat man das Problem, dass in der Klasse Bruch ja bereits die Methode **multipliziereMit** (s.o.) definiert ist. Glücklicherweise darf man in Java eine Methode mit gleichem Namen mehrfach definieren, wenn sich die Argumente unterscheiden. Man darf also der Klasse Bruch folgende Methode hinzufügen:

```
void multipliziereMit(Bruch faktor) {
    z = z * faktor.getZaehler();
    n = n * faktor.getNenner();
}
```

Mit einer solchen Bruch-Klasse darf nun sowohl die „elegante“ Methode als auch die Methode mit zwei Argumenten verwendent werden. Man sagt, die Methode **multipliziereMit** wurde überladen.

Diese Möglichkeit gibt es übrigens auch bei Konstruktoren: Zusätzlich könnten wir folgenden Konstruktor definieren:

```
Bruch(int ganzzahl, int zaehler, int nenner) {
    z = zaehler + nenner*ganzzahl;
    n = nenner;
}
```

Dieser unterscheidet sich vom „alten“ Konstruktor dadurch, dass jetzt auch gemischte Zahlen direkt als Bruchobjekt angelegt werden können (ohne im Hauptprogramm herumrechnen zu müssen)

Übung: Ergänze die Bruchklasse an allen sinnvollen Stellen mit entsprechenden überladenen Methoden

Vererbung

Wir wissen bereits, dass man in Java den gesamten Programmtext in Klassen organisiert. Von einer Klasse lassen sich (mit **new**) beliebig viele Objekte erzeugen. Es ist z.B. vorstellbar, dass man in einem Computerspiel eine Klasse "Gegenstand" anlegt:

```
class Gegenstand {
      String name;
      int gewicht;

      // hier fehlen nun ggf. ein Konstruktor und
      // Methoden wie "void setName(String n)"
      // oder "String getName()" usw.

}
```

Nun könnte man auf die Idee kommen, dass in dem Spiel auch spezielle "Kampfgegenstände" - z.B. ein Schwert oder Knüppel - vorkommen sollen. Diese Waffen sollen außer einem Namen und ein Gewicht (siehe Eigenschaften der Klasse Gegenstand) auch die Eigenschaft "Kampfkraft" bekommen, so dass das Programm unterscheiden kann, ob ein Schwert einem Knüppel im Spiel überlegen ist. Java kennt für diesen Fall das Konzept "Vererbung":

```
class Waffe extends Gegenstand {
      int kampfkraft;

      // hier fehlen noch Methoden, die nur für Waffen
      // interessant sind.
}
```

Die Klasse **Waffe** "erbt" nun die Methoden und Eigenschaften der Oberklasse (was sich an dem speziellen Wort **extends** erkennen lässt). In diesem Fall hat auch ein Objekt der Klasse **Waffe** die Eigenschaften **name** und **gewicht**, weil jedes Objekt der Klasse **Waffe** auch ein Objekt der Klasse **Gegenstand** ist (nicht aber umgekehrt!).

Java hat außerdem die Besonderheit, dass jede Klasse die Oberklasse **Object** besitzt (ohne dass man dies durch **extends** anzeigen muss). Damit kann man beliebige Objekte unterschiedlicher Klassen in einem Array zusammenfassen:

```
Object dinger = new Object[2];
dinger[0] = new Point(1,2);
dinger[1] = new JButton();
```

Man kann also in das oben definierte Array Objekte völlig unterschiedlichen Typs einspeichern. Holt man sie wieder dort heraus, so haben sie zunächst aber nur den Typ Object, so dass man den Datentyp zurückwandeln (casten) muss:

```
Point p = (Point) dinger[0]; // funktioniert
Point b = (Point) dinger[1]; // Exception: JButton ist kein Point
```

Vererbung: Methoden überschreiben

Wir betrachten folgende zwei Klassen:

```
class Ort {
      private String name;
      public void ausgabe() {
            System.out.println("Ortsname: "+name);
      }
}
```

Und eine passende Unterklasse:

```
class Hauptstadt extends Ort {
  private String land;
  public void ausgabe() {
    System.out.println("Die Stadt "+name);
    System.out.println("ist Hauptstadt von "+land);
  }
}
```

Es fällt auf, dass beide Klassen eine Methode mit dem Namen **`ausgabe()`** definieren. Normalerweise würde die Unterklasse **`Hauptstadt`** die Methode **`ausgabe()`** ja von der Klasse Ort erben. Aber sinnvollerweise sollte beim **`ausgabe()`**-Aufruf bei einem **`Hauptstadt`**-Objekt auch das Land mit ausgegeben werden. Daher wird in der Klasse **`Hauptstadt`** die Methode **`ausgabe()`** überschrieben. Die Methode gibt es nun sozusagen zweimal, wobei immer die passende Methode aufgerufen wird, je nachdem, welcher Art das Objekt ist.

Hat man beispielsweise ein Array definiert, in welchem **`Ort`**-Objekte gespeichert werden können, so darf man dort nun auch **`Hauptstadt`**-Objekte speichern, denn diese sind ja (spezielle) **`Ort`**-Objekte. Durchläuft man das Array und ruft bei jedem Element die **`ausgabe()`**-Methode auf, so wird die passende Methode gewählt.

Wir wissen bereits, dass jede Klasse die Oberklasse **`Object`** besitzt (ohne dass man dies durch **`extends`** anzeigen muss). Dadurch erbt jede Klasse auch die Methode **`toString()`**. Normalerweise gibt diese einen Hash-Wert, (eine Art Fingerabdruck des Objekts in der Gestalt **Klasse@6f30d50a**) zurück. Oft ist das aber nicht erwünscht, so dass man die Methode **`toString()`** überschreibt.

Insbesondere bei der GUI-Programmierung ist das Konzept "Vererbung" sehr praktisch: **`JButton`** , **`JLabel`** etc. sind Unterklassen von **`JComponent`** und erben damit Methoden wie z.B. **`setVisible(boolean v)`** (mit der man eine Komponente verstecken oder wieder sichtbar machen kann). Als Benutzer dieser Klassen ist uns dabei egal, ob diese Methode wirklich vererbt oder überschrieben wurde: Wir wissen nur, dass sie existiert – das genügt.

Interfaces

Das Wort „Interface“ bedeutet „Schnittstelle“. Eine Art von Schnittstelle in der Wirklichkeit ist z.B. ein Geldautomat: Man steckt die eigene Bankkarte in den Kartenschlitz, gibt die Geheimzahl und den gewünschten Geldbetrag ein und erhält schließlich das Geld. Ob der Geldautomat in der Fußgängerzone steht oder am Südpol von Pinguinen mit Solarstrom betrieben wird, ist für uns nicht wichtig: Wir erwarten nur, dass alles funktioniert.

Auf die Java-Programmierung übertragen bedeutet das: Unterschiedliche Klassen können eine gleichnamige Methode anbieten, die man immer auf die gleiche Art benutzt. In Java braucht man das Konzept oft in der GUI-Programmierung:

```
but1 = new JButton("Button 1");
but1.setBounds(10,10,200,30);
but1.addActionListener(this);
frame.add( but1 );
```

Wir schauen uns nun die dritte Zeile genauer an. Dort teilen wir der Java-Umgebung durch Aufruf von **`addActionListener`** mit, dass wir informiert werden möchten, falls der Button geklickt wird. Im Falle des Falles wird dann die Methode **`actionPerformed`** aufgerufen. Wie funktioniert das genau?

Die Methode **`addActionListener`** gehört zu **`JButton`** mit folgende Signatur:

```
public void addActionListener(ActionListener a)
```

Das übergebene Argument a muss also den Datentyp „**`ActionListener`**“ haben. „**`ActionListener`**“ ist aber keine normale bereits ausprogrammierte Klasse, sondern ein Interface, welches in der Java-Umgebung bereits festgelegt ist. In diesem Interface steht drin, dass jede Klasse, die ein „**`ActionListener`**“ sein möchte, die Methode „**`actionPerformed`**“ anbieten muss. In unserem obigen Beispiel übergeben wir aber das „**`this`**“-Objekt, welches z.B. den Typ „**`Startklasse`**“ hat. Damit nun alles funktioniert, muss die „**`Startklasse`**“ zwei Dinge tun (d.h. das Interface „**`ActionListener`**“ implementieren):

•A• im Klassenkopf muss „**`implements ActionListener`**“ stehen
•B• sie muss die Methode **`actionPerformed`** bereitstellen

Man kann sich ein Interface wie ein „Versprechen“ vorstellen, dass eine Klasse bestimmte Methoden auch wirklich anbietet.
Die Java-Umgebung will beim Button-Klick ja eine **`ActionPerformed`**-Methode aufrufen. Durch die beiden Dinge •A• und •B• haben wir sichergestellt, dass das immer klappt.

Wir können auch eigene Interfaces festlegen. In einem Computerspiel könnten z.B. sowohl Spielfiguren als auch computergesteuerte Geister oder Gasflaschen das Interface „Abschießbar“ implementieren. Die unterschiedlichen Klassen würden dann die Methode „abschießen()“ bereitstellen.

Datentyp char und Klasse String

Datentyp char: Wir kennen bereits Datentypen für Zahlen und Wahrheitswerte. Eine Variable vom Datentyp char kann genau einen Buchstaben aufnehmen (welcher in 'einfachen Hochkomma' angegeben ist - nicht wie bei der Klasse String mit "doppelten Gänsefüßchen"):

```
char c = 'H';       // OK
char s = 'ab';      // falsch! Nur ein Buchstabe erlaubt
char g = "H";   // falsch! "H" ist wegen doppelter
                           // Gaensefuesschen kein char
```

Klasse String: Mit Strings haben wir schon gearbeitet. Strings können aus unterschiedlich vielen Buchstaben (also Wörtern, ganzen Sätzen, ja sogar ganzen Büchern aber auch aus gar keinem Zeichen) bestehen. Strings sind hinter den Kulissen aus einzelnen char-Werten zusammengesetzt. Weil String eine Klasse ist, gibt es auch spezielle String-Methoden:

```
String papa = "Alfred";
int laenge = papa.length();    // Also 6
char dritter = papa.charAt(2); // f, von Null zählen
boolean b = papa.isEmpty();    // false
int idx = papa.indexOf('f');   // f an Position 2
```

Wir haben gelernt, dass man Objekte einer Klasse eigentlich anständig mit dem new-Operator „anlegen“ muss. Nur bei Strings gibt es die oben gezeigte Abkürzung. Eigentlich hätte man **papa** folgendermaßen definieren müssen:

```
String papa = new String(a); // mit a Array of chars
```

Aus technischen Gründen kann man die Zeichen eines String-Objektes nicht verändern. Die String-Klasse hat aber einen Zwilling, bei der das möglich ist:

Klasse StringBuffer: Objekte der Klasse StringBuffer muss man tatsächlich wie oben mit dem new-Operator anlegen. Dann aber stehen einige nützliche Methoden der Klasse zur Verfügung:

```
StringBuffer buf = new StringBuffer("Hallo");
int laenge = buf.length();    // Also 5
char zweiter = buf.charAt(1); // a, von Null zählen
buf.setCharAt(4,'i');         // jetzt "Halli"
buf.insert(3,'o');            // jetzt "Haloli"
buf.deleteCharAt(4);          // jetzt "Haloi"
String s = buf.substring(1,3); // Ergebnis "al"
```

Umwandlung von String in StringBuffer und umgekehrt:

```
String bla = "tralala";
StringBuffer blub = new StringBuffer(bla);
String bla2 = blub.toString();
```

Um ein char-Zeichen in einen String umzuwandeln, benutzen wir einen Trick:

```
String s = ""+'H'; // Leerstring mit angehängtem char
```

Java-Programme außerhalb von BlueJ

Bislang haben wir Java-Programme in der Entwicklungsumgebung „BlueJ“ erstellt. Eine solche Entwicklungsumgebung nennt man auch **IDE** (engl. für „Integrated Development Environment“). Es gibt viele weitere IDEs (z.B. „Eclipse“ oder „NetBeans“), die sich z.B. im Preis oder Funktionsumfang unterscheiden.
Ist ein eigenes Programm fertiggestellt, so soll es in der Regel auch für Nicht-Programmierer ohne Spezialkenntnisse ausführbar sein, d.h. unser Programm soll auch außerhalb von BlueJ (oder einer anderen IDE) funktionieren. Dazu sind folgende Schritte notwendig.

Erstens: Zunächst braucht ein Java-Programm einen „Anfang“, d.h. es muss irgendwie festgelegt sein, in welcher Klasse in welcher Methode die Befehlsausführung beginnt, wenn das Programm gestartet wird. Dazu gibt es folgende Konvention: In einer Klasse, die als „Startklasse“ fungiert, muss eine Methode mit dem Namen **`main`** existieren:

```
class MeineStartklasse {
      public static void main(String[] args) {
      // .... hier geht es los ....
      }
}
```

Zweitens: Wenn das Programm in BlueJ compiliert und getestet wurde, kann man BlueJ nun schließen. Nun sucht man im Dateisystem den Ordner mit den Projektdateien. Wenn korrekt compiliert wurde, so müsste sich dort u.a. die Datei **`MeineStartklasse.java`** und **`meineStartklasse.class`** befinden. Eigentlich ist jetzt nur noch die **`.class`**-Datei wichtig, denn darin befindet sich der übersetzte Programmcode, der für den Computer (aber nicht mehr für uns) lesbar ist. Ein Doppelklick auf diese Datei startet aber noch nicht das Programm. Wir erstellen daher eine sogenannte **Batch-Datei**, welche eine Anweisung an das Windows-Betriebssystem enthält, unser Java-Programm auszuführen: Starte also den Editor Notepad und speichere folgende zwei Zeilen unter dem Namen **`start.bat`**:

```
java MeineStartklasse
pause
```

Liegt diese Batchdatei nun im gleichen Verzeichnis wie unsere **`.class`**-Datei, so wird beim Doppelklick auf die Batch-Datei das Java-Programm gestartet.

Eingaben im Textmodus: Manchmal ist es sinnvoll, Eingaben nicht über einen JDialog vom Benutzer zu erbitten, sondern direkt über eine Eingabe in der Textzeile. Dies klappt mit folgender Java-Zeile:

```
String eingabe = System.console().readLine();
System.out.println("Die Eingabe war "+eingabe);
```

Achtung: Diese Methode funktioniert nicht innerhalb von BlueJ!

Der Punkt-Operator

Um eine Methode aufzurufen, sind wir bislang folgendermaßen vorgegangen:

```
Bruch meinBruch = new Bruch(2,3); // Objekt anlegen
Bruch deinBruch = new Bruch(5,6); // Objekt anlegen
Bruch ergebnis = new Bruch(meinBruch);
ergebnis.multipliziere(deinBruch);
```

Hinter dem Punkt steht ein Methodenname, vor dem Punkt ein Objekt (oder ein Klassenname - siehe weiter unten).
Wenn die im Beispiel benutzte **multipliziere**-Methode nichts anderes tut, als das Objekt zu verändern, auf dem sie angewendet wurde (im Beispiel hatte **ergebnis** erst den Wert von **meinBruch**, nach **multipliziere** den Wert des Ergebnisses), werden Rechnungen unangenehm unübersichtlich.
Eleganter wäre: **ergebnis = meinBruch · deinBruch**

Leider geht das nicht so ohne Weiteres. Aber dass die Methode **multipliziere** das „eigene“ Objekt verändert, ist eine unschöne Designentscheidung. Viel eleganter ist folgende Lösung:

```
public Bruch multipliziere2(Bruch b) {
      int erg_z = z * b.getZaehler();
      int erg_n = n * b.getNenner();
      return new Bruch(erg_z,erg_n);
}
```

Diese neue Methode verändert nicht die Eigenschaften **z** und **n** des „eigenen“ Objektes. Stattdessen wird ein neues Ergebnisobjekt erstellt und als Rückgabewert zurückgegeben. Damit ist nun folgender Aufruf möglich:

```
Bruch meinBruch = new Bruch(2,3); // Objekt anlegen
Bruch deinBruch = new Bruch(5,6); // Objekt anlegen
Bruch ergebnis = meinBruch.multipliziere2(deinBruch);
```

Und was noch viel besser ist: Aufrufe dieser Art kann man verketten! Mit entsprechenden Methoden ist nun folgendes möglich:

```
Bruch d = a.addiere2(b).multipliziere2(c);
```

Hier wird also zunächst a + b gerechnet. Da das Ergebnis wieder ein Bruch-Objekt ist, kann darauf auch wieder eine Methode angewandt werden. Im Objekt d ist nun also das Ergebnis der Rechnung (a+b) * c enthalten!

Statische Methoden

Bestimmte Methoden können aufgerufen werden, obwohl kein Objekt der entsprechenden Klasse existiert. Z.B. ist die Methode **ggT** in der Bruch-Klasse auch ohne konkretes Bruch-Objekt sinnvoll. Deswegen taucht bei der Methodendefinition das Wort **static** auf. Um diese Methode von außen ohne existierendes Bruch-Objekt aufzurufen, schreibt man:

```
int c = Bruch.ggt(6,8);
```

Achtung: Statische Methoden dürfen nicht auf Objekteigenschaften zugreifen!