Einführung in Java

Chris Weber, Kantonsschule Limmattal

EF Informatik 2023/24

1 Imperative und funktionale Programmierung

1.1 Hello World (Github, IDE, Terminal, erstes Programm)

1. Gehe auf https://github.com/KS-Limmattal/EF-Informatik-2022-23/ und nimm die Aufgabe 1 an. Klone den Code auf deinen Computer (bzw. Programmier-Stick), z.B. in einen Ordner `Projects`:

   - Arbeitest du auf einem eigenen Computer, solltest du Git installieren (https://git-scm.com/download). Dann kannst du in Visual Studio Code (VSC) auf Version Control klicken (drittoberstes Icon am linken Rand) und danach auf „Clone Repository“. Du musst dich in deinen Github-Account einloggen, danach kannst du dein Repository klonen.

   - Arbeitest du mit Programmierstick auf einem Schulcomputer, musst du leider das Repository von der Webseite herunterladen und entzippen, da auf den Schulcomputern kein Git läuft.

2. Du solltest nun einen Unterordner `23_01a_Grundlagen` haben, in dem bereits einige Dateien liegen. Diese kannst du im Moment ignorieren, du wirst sie in Abschnitt 1.3 brauchen. Stelle sicher, dass du den Unterordner `23_01a_Grundlagen` in VSC geöffnet hast (ggf. mit Ctrl&K, Ctrl&O öffnen).

3. Erstelle in VSC eine neue Datei mit dem Namen `Hello.java` (Gross- und Kleinschreibung beachten!) und fülle sie mit dem folgenden Code (Tabulator für Einrückungen):

```java
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

   Speichere (Ctrl&S).

4. Öffne eine **Konsole** (Terminal/ New Terminal) und tippe darin die folgenden Befehle (jeweils mit Enter bestätigen):

   `javac Hello.java` (**Kompiliert** die Datei zu `Hello.class`)

   `java Hello` (führt die Datei `Hello.class` aus)

Kontrolliere mit `ls` oder `dir` (je nach Betriebssystem), welche Dateien im Verzeichnis sind.

weiterer Grundbefehl: `cd ..` bzw. `cd 01_Grundlagen` (*change directory*)

5. Eine „Vorschau“, in der die Schritte aus dem vorherigen Punkt im Hintergrund ausgeführt werden, ist der „Run“-Knopf, der über der `main()`-Methode erscheint (kann auch über die Taste F5 aufgerufen werden). Lösche die Datei `Hello.class` und probiere ihn aus. Was fällt auf?

   Antwort: Es wird keine neue Datei `Hello.class` erstellt! Die kompilierte Klasse wird also lediglich temporär im Arbeitsspeicher erstellt.

6. Mache auf Github einen Commit, der deine Datei `Hello.java` enthält:

   - Arbeitest du auf einem eigenen Computer, kannst du in VSC/Version Control „Commit“ (in dein lokales Repository) und danach „Sync Changes“ verwenden.
   - Arbeitest du auf einem Schulcomputer, lade die Datei direkt auf der Webseite hoch (im Repository unter „Add File“).

In unserem ersten Programm stehen sehr viele kryptische Schlüsselwörter, deren Bedeutung wir nach und nach kennenlernen werden. Ein paar erste Erklärungen:

- Die **Klasse** `Hello` enthält unser „Hello World“-Programm.
- die **Methode** `main()` wird ausgeführt, wenn `Hello` ausgeführt wird. Klassen können noch weitere Methoden enthalten.
- `System` ist eine bereits bestehende Klasse, auf die wir in unserem Programm zugreifen. Sie enthält ein Objekt `out`, das die Methode `println()` enthält, die einen übergebenen String (Text) auf der Konsole ausgibt.

1.2 Calc (Variablen, Datentypen int und String, Operationen, Syntax)

1. Erstelle eine Datei `Calc.java` mit dem Inhalt

```java
public class Calc {
  public static void main(String[] args) {
    int a = 34;
    int b = 7;
    System.out.println("sum=" + a + b);
  }
}
```

   Was erwartest du, dass das Programm macht? Versuche eine Theorie aufzustellen, was die Code-Zeilen bedeuten. Probiere es danach aus (kompilieren und ausführen im Terminal oder über den „Run“-Knopf). Hast du richtig geraten? Falls nein, hast du eine Erklärung dafür?

Antwort:
`int a = 34;` macht drei Dinge: Es definiert eine **Variable** mit dem Namen `a`, definiert, dass sie vom Typ `int` (Ganzzahl, *integer*) sein soll, und speichert darin den Wert 34.
Naiv könnte man erwarten, dass das Programm die Summe aus `a` und `b` berechnet und danach `sum=41` auf der Konsole ausgibt. Das ist aber nicht der Fall.
`"sum="` ist Text, also vom Datentyp `String`. Wenn Java den Ausdruck `"sum=" + a` sieht, wandelt es den Inhalt von `a` in einen String um (das nennt man **automatische Typkonversion**) und hängt die beiden Strings zu einem neuen String `"sum=34"` zusammen. Danach passiert das Gleiche mit dem String `"sum=34"` und dem Inhalt von `b`, so dass am Schluss der String `"sum=347"` herauskommt.

2. Versuche, als Resultat die Summe von a und b angezeigt zu bekommen.

Mögliche Lösungen:
- Eine dritte Variable `int c = a + b;` definieren und danach `"sum=" + c` ausgeben.
- Klammern setzen: `System.out.println("sum=" + (a + b));`. Java verrechnet hier (mathematisch gut erzogen) zuerst die Dinge in der Klammer miteinander.

In beiden Fällen erreichen wir, dass die Operation `a + b` ausgeführt wird. Da `a` und `b` beides Zahlen vom Typ `int` sind, bedeutet `+` hier die ganz normale ganzzahlige Addition. Danach wird das Resultat (ebenfalls eine `int`-Zahl) mit der `String`-Operation `+` an `"sum="` angehängt.

3. Ersetze `+` der Reihe nach durch `-`, `*`, `/` und `%` und beobachte das Verhalten. Falls nötig, experimentiere mit anderen Werten für `a` und `b`, um zu verstehen, was passiert.

4. Sind die Einrückungen, Zeilenwechsel und Semikolons nötig für einen reibungslosen Ablauf des Programms? Probiere aus!

Antwort: Die Zeilenwechsel und Einrückungen haben keinen Einfluss auf die Funktion des Programms. Sie sind lediglich für uns Menschen da, um die Übersicht nicht zu verlieren. Das heisst, wir können auch in überlange Befehlszeilen Zeilenwechsel einfügen, so dass alles auf einer Bildschirmbreite Platz hat. Für den Java-Compiler ist das Semikolon das Zeichen für einen abgeschlossenen Befehl. Es ist deshalb unverzichtbar.

5. Unter https://javabeginners.de/Grundlagen/Datentypen/Primitive_Datentypen.php findest du eine Übersicht über die **primitiven Datentypen**.

1.3 Powers (Methoden, Kommentare, boolean, Verzweigungen)

1. Studiere nun die Klasse `Powers.java`, die im geklonten Repository schon vorhanden war. Was verstehst du? Führe sie aus und überprüfe deine Überlegungen.

2. Ein paar Erläuterungen:

   - In der Datei `Powers.java` ruft die `main()`-Methode die Methode `square()` auf. Dabei übergibt sie den Integer `base` als Argument und bekommt als `return`-Wert wieder einen Integerwert zurück. Deshalb muss `int` (=der Typ des Rückgabewertes) vor der Definition der Methode `square()` stehen. Wir sagen auch, die Methode `square()` sei **vom Typ** `int`. Wenn eine Methode nichts zurückgibt (wie die `main()`-Methode), steht `void`.
   - `static` muss im Moment vor jeder Methode stehen, wir werden später sehen, wieso.
   - `//` und der `/** **/`-Block markieren **Kommentare** (ein- bzw. mehrzeilig), die vom Compiler ignoriert werden. Der `/** * **/`-Block ist in diesem Beispiel ein **Javadoc**-Kommentar. Er ist so formatiert, dass daraus automatisch Dokumentationsseiten für unser Java-Projekt erstellt werden könnten. Auch die Klassen aus der Java-Library sind auf diese Art dokumentiert: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/System.html Finde in der Dokumentation zur Klasse `System` die Beschreibung der Methode `println()`. Was bemerkst du?

     Antwort:
     - `out` ist eine Variable vom Typ `PrintStream`. Deshalb besitzt es auch die Methoden dieses Typs.
     - Es gibt nicht nur eine Methode `println()`, sondern deren zehn! Jede davon nimmt einen anderen Datentyp entgegen. Es ist in Java tatsächlich möglich, Methoden zu **überladen** (***method overloading***), d.h. mehrere Methoden mit dem gleichen Namen, aber unterschiedlicher Zahl oder Typ von Argumenten zu definieren.
     - `println()` ohne Argument gibt nur einen Zeilenwechsel aus. Die anderen Methoden sind aus `print()` mit dem gleichen Argumenttyp und einem Zeilenwechsel (argumentloses `println()`) zusammengesetzt.

3. Ergänze die Datei mit einer zweiten Methode `cube()`, die die dritte Potenz eines `int`-Arguments `base` zurückgibt. Teste sie mit einer Ausgabe in der Konsole.

4. Berechne `cube(10000)`. Hast du eine Idee, was da passiert? Probiere aus!

   Antwort: Offenbar gibt Ihre Methode für Argumente grösser als 8470 Quatsch aus. Das hat damit zu tun, wie `int`-Zahlen im Speicher dargestellt und verarbeitet werden. Mehr Details mündlich!
   Fortgeschrittenen-Aufgabe: Erkläre möglichst genau, warum und wie das falsche Resultat zustande kommt.

5. Wir wollen eine Warnung auf der Konsole ausgeben, falls `base` grösser als 8470 ist. Ergänze vor dem `return`-Befehl in der Methode `cube()` den Code

```
if (base > 8470) {
  System.out.println("Warning: The cube of " + base + " is outside the
     range of int.");
}
```

Probiere aus.

6. Ersetze die Klammer `}` des `if`-Befehls durch

```
} else if (base == 0 || base == 1) {
  System.out.println("useless operation, but ok");
} else {
  System.out.println("Nothing to worry about :-)");
}
```

und probiere aus. Du hast eine **Verzweigung** programmiert.

7. Die Bedingung `(...)` für das Ausführen der geschweiften Klammer ist vom Typ `boolean` (mögliche Werte: `true` oder `false`). Boolesche Werte können in Variablen gespeichert werden mittels z.B. `boolean a = true;` oder `boolean bedingung = (a >= 0);` Sie können dann benutzt werden in Ausdrücken wie `if a {...}`. Experimentiere damit!

8. **Boolesche Logik:**

`&&`	und (AND)	`==`	ist gleich	`<`	ist kleiner als
`||`	oder (OR)	`!=`	ist ungleich	`>=`	ist grösser/ gleich
`!a`	nicht (NOT) `a`	`>`	ist grösser als	`<=`	ist kleiner/ gleich

Ausserdem kann mit Klammern gearbeitet werden:

```
(a > b && !(a <= 0)) || a == 0
```

Es gelten die **Regeln von DeMorgan:**

```
!(a && b) = !a || !b
!(a || b) = !a && !b
```

9. Betrachte die Datei `Variablensichtbarkeit.java`. Wenn du in den kommentierten Stellen `// (1)` bis `// (5)` den Wert der Variablen `a` und `b` ausgibst, was erwartest du? Probiere es danach aus! Entsprechen die Resultate deinen Theorien? Wenn nicht, hast du Erklärungen dafür?

Erklärung:
Das Programm durchläuft die Kommentare in der angegebenen Reihenfolge. Es gibt zwei unterschiedliche Variablen `a`, nämlich in jeder der beiden Methoden eine. Jede Methode hat nur Zugriff auf die eigene **lokale Variable** `a`. Zum Zeitpunkt `// (3)` ist `a` undefiniert.
`b` hingegen ist eine **globale Variable**, die in der ganzen Klasse definiert ist und von überall her verändert und zugegriffen werden kann.

10. Weisst du, was im Speicher geschieht, wenn `Variablensichtbarkeit` ausgeführt wird?

11. Bearbeite die Übungen 1 und 2 in `23_01a_Grundlagen` und lade deine Lösungen auf Github hoch.

1.4 Binäre Codierung ganzer Zahlen

Unter https://javabeginners.de/Grundlagen/Datentypen/Primitive_Datentypen.php findest du eine Übersicht über die **primitiven Datentypen**. Wir betrachten im Folgenden die Codierung von Ganzzahlen in den Typen `byte`, `short`, `int` und `long` genauer. Die vier Datentypen unterscheiden sich lediglich darin, wie viele Bits für die Speicherung einer Zahl zur Verfügung stehen.

Definition 1. Stellenwertsysteme zur Darstellung von Zahlen (z.B. Binär-, Dezimal-, Hexadezimalsystem) funktionieren wie folgt: Zu einer **Basis** b und einer Menge von **Ziffern** $Z = \{0, 1, \dots b-1\}$ werden Zahlen mit Ziffern $z_i \in Z$ wie folgt dargestellt:

$$z_n z_{n-1} \dots z_1 z_0 = z_n \cdot b^n + z_{n-1} \cdot b^{n-1} + \dots + z_1 \cdot b^1 + z_0 \cdot b^0$$

Beispiel 1. Die wichtigsten Systeme für die Informatik sind:

- **Dezimalsystem**: $243_{10} = 2 \cdot 10^2 + 4 \cdot 10^1 + 3 \cdot 10^0$
- **Binärsystem**: $1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11_{10}$
- **Hexadezimalsystem**: $\text{A69F}_{16} = 10 \cdot 16^3 + 6 \cdot 16^2 + 9 \cdot 16^1 + 15 \cdot 16^0 = 42\ 655_{10} = 1010\ 0110\ 1001\ 1111_2$. Beachte, dass jeweils ein Viererblock im Binärsystem einer Ziffer im Hexadezimalsystem entspricht, da $2^4 = 16$. Deshalb wird das Hexadezimalsystem verwendet, um längere Bitfolgen anschaulicher darzustellen.

Aufgabe 1. Wandle um:

1. vom Dezimal- ins Binär- und Hexadezimalsystem:

   a) 10 b) 100 c) 256 d) 2023

2. vom Binär- ins Dezimal- und Hexadezimalsystem:

   a) 10 b) 1 0110 c) 0110 0110 d) 11111010101

3. vom Hexadezimal- ins Binär- und Dezimalsystem:

   a) 10 b) D2 c) AFFE

Lösungen:

1. a) $1010_2 = A_{16}$ b) $110\ 0100_2 = 64_{16}$ c) $1\ 0000\ 0000_2 = 100_{16}$ d) $111\ 1110\ 0111_2 = 7E7_{16}$
2. a) $2_{10} = 2_{16}$ b) $22_{10} = 16_{16}$ c) $102_{10} = 66_{16}$ d) $2005_{10} = 7D5_{16}$
3. a) $1\ 0000_2 = 16_{10}$ b) $1101\ 0010_2 = 210_{10}$ c) $1010\ 1111\ 1111\ 1110_2 = 45\ 054_{10}$

Definition 2. Ganzzahlige Datentypen werden im **Zweierkomplement (*two's complement*)** dargestellt: Liegt die binäre Codierung einer natürlichen Zahl n in der Anzahl Bits des Datentyps vor, so erhält man die binäre Codierung von $-n$ wie folgt:

1. Invertiere jedes Bit der Codierung von n
2. Addiere 1 dazu (wobei zusätzliche Bits abgeschnitten werden)

Beispiel 2. Sei 0000 0110 die binäre Codierung von $n = 6$ als `byte` (=8 Bit).

1. Invertiere: 1111 1001
2. Addiere 1: 1111 1010 $= -6_{10}$

Aufgabe 2. Rechnen im Zweierkomplement:

1. Codiere als `byte` im Zweierkomplement:

   a) -24 b) -0 c) $-(-6)$

2. Was ist die grösste und kleinste mögliche Zahl, die im Zweierkomplement in 8 Bit codiert werden können?
3. Berechne schriftlich im Binärsystem als `byte`. Behandle dabei Subtraktionen als Addition der Gegenzahl:

   a) $12 + 23$ c) $105 - 234$ e) $66/3$

   b) $8 - 5$ d) $5 \cdot 111$

1.5 Fortsetzung von Powers (Schleifen, Debuggen, Stack)

Nimm die Aufgabe 2 auf https://github.com/KS-Limmattal/EF-Informatik-2022-23/ an.

1. Ergänze die Klasse `Powers` mit der folgenden Methode:

```
static int power(int base, int n) {
  int power = 1;
  int i = 0; //counter
  while (i < n) { //repeats what is in {} as long as () is true
    power = power * base;
    i++; //short for "i = i + 1";
  }
  return power;
}
```

   Schreibe wieder Testcode, um `power(a, b)` zu testen.

2. Der obige Code lässt sich auch kürzer schreiben, nämlich mit einer `for`-Schleife. Der folgende Code ist für den Compiler identisch. Vergleiche die beiden Code-Schnipsel und behalte eines!

```
static int power(int base, int n) {
  int power = 1;
  for (int i = 0; i < n; i++) {
    power = power * base;
  }
  return power;
}
```

3. Nehmen wir an, `power()` wurde in `main()` mit

```
1 int a = power(2, 3);
```

aufgerufen. Stelle bildlich dar, was im Stack zu welchem Zeitpunkt gespeichert ist:

- Nach der Initialisierung der Schleife (Z. 4 bzw. 3).
- Jedes Mal, wenn das Ende des Schleifenrumpfs erreicht wird (Z. 7 bzw. 5)
- Nachdem `power()` fertig ausgeführt wurde und der `return`-Wert an die `main()`-Methode zurückgegeben wurde.

Tipp: Der Debugger kann helfen, deine Überlegungen teilweise zu überprüfen. Setze Haltepunkte (rote Punkte links neben den Zeilennummern) an den genannten Stellen. Führe danach das Programm mit „Debug“ (oberhalb der `main()`-Methode) aus und beobachte die Variablen links oben im Debug-Bereich von VSC.

4. Betrachte die Klasse `PerfectSquareCounter.java`. Sie soll für eine gegebene Zahl n die Anzahl Quadratzahlen sowie die Anzahl der geraden Quadratzahlen $\leq n$ berechnen und ausgeben. Die Klasse ist nicht wahnsinnig elegant programmiert, aber sie kompiliert und gibt keine Fehler aus. Allerdings zeigt sie nicht das gewünschte Verhalten.

   Finde alle Fehler im Code (und behebe sie) mit Hilfe des Debuggers. Werte globaler Variablen (und übrigens auch Werte komplexerer Ausdrücke) kannst du im Bereich „Watch“ beobachten, indem du einen Ausdruck mit dem Namen der Variablen hinzufügst. Experimentiere wenn nötig mit unterschiedlichem Testcode.

Antwort:

- Die Methode `calculateCount()` gibt die falsche Zahl zurück, nämlich den Counter für die geraden Quadratzahlen (Zeile 44).
- In Zeile 39 steht eine Bedingung für *ungerade* Zahlen statt für gerade, so dass `evenPerfectSquareNumbers` genau die falschen Zahlen zählt.
- Irgendwie hat sich ein Minus eingeschlichen in Zeile 36, so dass -1 und 0 ebenfalls geprüft werden. Da 0 ebenfalls als Quadratzahl gezählt wird, wird das Ergebnis verfälscht.
- In der gleichen Zeile ist die Abbruchbedingung `number < i` falsch. Damit wird i selbst nicht mehr mitgeprüft. Dieser Fehler könnte durch automatisiertes Kopieren des Mustersi `from (i = 0; i < n; i++)` entstanden sein, wo tatsächlich n Zahlen durchlaufen werden. Allerdings beginnt hier die Zählung sinnvollerweise bei 1 statt 0, so dass es sich um einen **off-by-one error** handelt.

1.6 Fliesskommazahlen (float, double)

Aufgabe 3. Betrachte die Klasse `Double.java` und führe sie aus.

1. Probiere aus, was herauskommt, wenn du das `+` durch ein `-`, `*`, `/` oder `%` ersetzt.
2. Mache den Code von Teilaufgabe 2 ausführbar, indem du die Kommentarbalken entfernst (der Übersicht halber kannst gleichzeitig den Code von Teilaufgabe 1 auskommentieren). Kannst du den Code so verändern, dass `0.5` ausgegeben wird? Suche einfache Lösungen!

Wenn du eine gefunden hast, probiere aus, ob es noch kürzer geht.

Das Problem ist, dass `1` und `2` als `int` interpretiert werden. Das kann auf verschiedene Arten geändert werden:

- `1.0 / 2.0` (oder auch nur `1. / 2.`) macht klar, dass Fliesskommazahlen gemeint sind.
- Alternativ funktioniert auch der Zusatz `d` für `double`: `1d / 2d`
- Es reicht sogar, dass eine der Zahlen ein `double` ist. Dann wird die Operation automatisch eine `double`-Operation, und die andere Zahl wird automatisch in eine `double`-Zahl konvertiert. Es funktionieren also z.B. auch `1. / 2` oder `1 / 2d`.
- Eine manuelle **Typkonversion** ***(cast)*** einer `int`-Zahl `a` kann mit `(double)` `a` erreicht werden.

Vorsicht: `(double) 1 / 2` funktioniert nicht, da sich die Typkonversation auf das *Ergebnis* der `int`-Division bezieht. `((double) 1) / 2` läuft hingegen.

3. Führe den Code von Teilaufgabe 3 aus. Hast du eine Erklärung für das Verhalten?

Das Problem ist, dass die Zahlen nicht als Dezimal-, sondern als Binärbruch dargestellt werden. Die nachfolgenden Definitionen und Aufgaben versuchen, Licht ins Dunkel zu bringen.

Definition 3. Fliesskommazahlen werden zur Speicherung als `float` oder `double` zunächst analog zur dezimalen wissenschaftlichen Schreibweise (z.B. $2.71828 \cdot 10^{-23}$) geschrieben - einfach binär statt dezimal:

$$(-1)^S \cdot m \cdot 2^e,$$

wobei S das Vorzeichenbit ist, m die Mantisse (die immer mit einer 1 gefolgt von einem Dezimalpunkt beginnt) und e der Exponent (alles Binärzahlen!).

Davon werden gespeichert:

- S (1 bit)
- $E = e + B$ für einen Biaswert B, so dass E zu einer positiven Ganzzahl wird.
  Für `float` hat E 8 Bits, und $B = 2^{8-1} - 1 = 127$.
  Für `double` hat E 11 Bits, und $B = 2^{11-1} - 1 = 1023$.

  Spezialfall: Für den kleinst- und grösstmöglichen Wert von E (also entwerder lauter Nullen oder lauter Einsen) gilt das Codierungsschema nicht. Damit werden Werte wie 0, sehr kleine Zahlen, unendlich und `NaN` („*not a number*“) codiert.[1]
- $M = (m \cdot 2^p)\%(2^p)$, wobei p die Anzahl Bits von M ist. Einfacher gesagt: M ist m ohne die führende 1 und den Dezimalpunkt (also ebenfalls eine positive Ganzzahl).
  Für `float` ist $p = 23$, für `double` $p = 52$.

[1] Mehr Informationen siehe https://de.wikipedia.org/wiki/IEEE_754.

Beispiel 3. Der Dezimalbruch 18.4 soll als `float` gespeichert werden.

1. Als Bruch schreiben, kürzen, in Binärbruch umwandeln: $18.4 = \frac{184}{10} = \frac{92}{5} = \frac{101\ 1100_2}{101_2}$
2. Schriftliche Division:

$$\begin{array}{r} 101\ 1100 : 101 = 1\ 0010.\ 0110\ 0110... \\ \underline{-101} \\ 0\ 110 \\ \underline{-\ 101} \\ 10\ 00 \\ \underline{-\ 1\ 01} \\ 110 \\ \underline{-\ 101} \\ \vdots \end{array}$$

3. Normalisieren zu $(-1)^0 \cdot 1.\ 0010\ 0110\ 0110... \cdot (2^4)_{10} \overset{\text{binär}}{=} (-1)^0 \cdot 1.\ 0010\ 0110\ 0110... \cdot 10^{100}$
4. $S = 0$
5. Bestimmen von E durch Addieren der Bias zu $e = 4_{10} = 100$:
   $E = e + 127_{10} = e + 111\ 1111 = 1000\ 0011 = 131_{10}$
6. Ignorieren der Vorkommastelle der Mantisse, 23 Nachkommastellen mathematisch runden: $M = 001\ 0011\ 0011\ 0011\ 0011\ 0011$ (abgerundet, da danach eine 0 folgt)
7. Die Bitfolgen für das Vorzeichen $S = 0$, den Exponenten $E = 1000\ 0011$ und die Mantisse $M = 001\ 0011\ 0011\ 0011\ 0011\ 0011$ werden verkettet: 0100 0001 1001 0011 0011 0011 0011 0011.

Aufgabe 4. Wandle die `float`-Zahl 0100 0001 1001 0011 0011 0011 0011 0011 wieder in einen Dezimalbruch um.

Aufgabe 5. Bestimme die grösste und die kleinste positive Zahl, die als `float` bzw. `double` nach dem Standardschema gespeichert werden kann

Lösung: Für positive Zahlen gilt $S = 0$, was aber für die Fragestellung irrelevant ist. E kann nicht nur aus Nullen oder nur aus Einsen bestehen, da diese „Exponenten" ja für Spezialcodierungen reserviert sind. Deshalb werden diese Werte jeweils um 1 erhöht bzw. vermindert.

- Für die kleinste Zahl ist $E = 1$ und $M = 0$, also $m = 1$:
  - In `float` hat die Zahl $e = -127$, ist also gleich $1 \cdot 2^{-127} \approx 5.88 \cdot 10^{-39}$.
  - In `double` hat die Zahl $e = -1023$, ist also gleich $1 \cdot 2^{-1023} \approx 1.11 \cdot 10^{-308}$.
- Für die grösste Zahl gilt:
  - In `float` ist $E = 1111\ 1110 = 254_{10}$, also $e = 254 - 127 = 127$,
    $M = 2^{23} - 1$, also $m = \frac{2^{24}-1}{2^{23}} = 2 - 2^{-23}$
    und somit die grösste Zahl $(2 - 2^{-23}) \cdot 2^{127} \approx 3.40 \cdot 10^{38}$.
  - In `double` ist $E = 111\ 1111\ 1110 = 2046_{10}$, also $e = 2046 - 1023 = 1023$
    $M = 2^{52} - 1$, also $m = \frac{2^{53}-1}{2^{52}} = 2 - 2^{-52}$
    und somit die grösste Zahl $(2 - 2^{-52}) \cdot 2^{1023} \approx 1.80 \cdot 10^{308}$.

Aufgabe 6. Versuche möglichst genau zu erklären, was bei der `double`-Operation `0.1 + 0.2` geschieht!

Hinweise:

$0.\ 0100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 11$

$0.\ 0100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1101\ 00$

$0.\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 010$

$0.\ 0001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1010$

Aufgabe 7. Code lesen

1. Das folgende Programmstück vertauscht den Inhalt der Variablen `x` und `y`, falls `x` grösser als `y` ist. Stimmt das?

```
int x = 2; y = 1;
if (x > y)
  int swap = x;
  x = y;
  y = x;
// x sollte 1 sein und y 2
```

2. Wie viele Sternchen werden bei den folgenden Schleifen auf der Konsole erscheinen?

```
// Programm A:
for (int stars = 0; stars <= 7; stars = stars + 2){
  System.out.println("***");
}
```

```
// Programm B:
for (int stars = 10; stars < 0; stars++){
  System.out.println("**");
}
```

Aufgabe 8. Löse die Aufgaben in `Aufgabe8.java` und gib deine Lösungen auf Github ab. **Bemerkung** zu Teilaufgabe c): Die Kreiszahl π kann mit der **Madhava-Leibniz-Reihe** (entdeckt im 14. Jh. in der Schule von Madhava von Sangamagrama und 1673 nochmals von Gottfried Wilhelm Leibniz) berechnet werden:

$$\pi = 4 \cdot \lim_{n\to\infty} \sum_{i=0}^{n} \frac{(-1)^i}{2i+1} = 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots\right)$$

Wird die n-te Partialsumme berechnet, kann $4 \cdot \frac{1}{n}$ als obere Schranke für den Fehler verwendet werden.

1.7 Arrays

Klone das Github-Projekt `03_Arrays`. Studiere die Methoden `serialHello1()` und `main()` in der Klasse `SerialHello.java` und führe ihren Code aus.

Erklärungen: `String[] names` definiert ein **Array** von `Strings`, d.h. eine nummerierte Menge von `Strings`. Seine Elemente werden mit `names[0]` bis `names[5]` angesteuert, wobei die Zahlen 0 bis 5 **Indizes (Sg. Index)** heissen. Es hat eine Eigenschaft `length` vom Typ `int` (und Wert 6), die angibt, wie viele String-Speicherplätze es enthält.

Aufgabe 9. Nachdem die Kursliste schon gedruckt und das Java-Programm geschrieben ist, wird ein Repetent zum Kurs hinzugefügt. Lasse auch ihn begrüssen, indem du seinen Namen zum Array hinzufügst.

Studiere nun die Methode `serialHello2()`. Sorge dafür, dass sie in Z. 16 statt `serialHello1()` aufgerufen wird und führe das Programm aus. Welche Vor- und Nachteile hat diese **„for each“**-Notation gegenüber der von `SerialHello1()`?

Mögliche Antworten:
Vorteile:
- Der Code hat weniger Potential für (u.U. schwer auffindbare) Fehler.
- Der Code wird kürzer und ist klarer lesbar.
- Man muss sich nicht mit Zählern und Indizes herumschlagen und ist somit schneller beim Programmieren.

Nachteile:
- Die Notation kann nur verwendet werden, wenn alle Elemente des Arrays durchlaufen werden sollen.
- Wir kennen die Indizes der Elemente, die wir durchlaufen, nicht. Es ist damit also nicht möglich, direkt auf Nachbarn von Elementen zuzugreifen.
- Bei Arrays haben wir Glück, dass die Reihenfolge der Elemente beibehalten wird. Wenn wir andere Objekte mit einer "for each"-Schleife durchlaufen (`List`, `Set`, `Heap` etc.), ist das nicht unbedingt garantiert.
- Wertänderungen wie `for (int i : intArray){ i = 1; }` sind nicht möglich.

Aufgabe 10. Das obligatorische Argument `args` der `main`-Methode ist ebenfalls ein `String`-Array. Wird die Klasse „von Hand“ auf der Kommandozeile mit `javac` kompiliert und mit `java` ausgeführt, dann werden durch Leerschläge getrennte Strings nach dem Klassennamen als Elemente in das Array eingelesen. Konkret: `java SerialHello dies das "und jenes"` führt zu `args = {"dies", "das", "und jenes"}`. Benutze diese Information, um das Programm so umzuschreiben, dass der Kommandozeilen-Aufruf `SerialHello Alice Bob Mallory` (nach Kompilation mit `javac SerialHello.java`) die Ausgabe

```
Hello Alice!
Hello Bob!
Hello Mallory!
```

ergibt.

Lösung: In Zeile 16 `names` durch `args` ersetzen. Z. 15 wird dann obsolet.

Syntax

Arrays können auch von beliebigen anderen Datentypen erstellt werden, z.B. `int` oder `double` oder von anderen nicht-primitiven Datentypen. Die Syntax aus der folgenden Tabelle (am Beispiel eines `int`-Arrays) kannst du in `ArrayTests.java` ausprobieren. Experimentiere, bis du dich damit sicher fühlst!

`int[] a;`	Deklaration des Arrays a als `int`-Array
`a = new int[3];`	Erzeugung eines `int`-Arrays der Grösse 3 automatische Initialisierung mit Inhalt {0, 0, 0} Zuweisung des Arrays zur Variablen `a`.
`a[0] = 5;` `a[2] = 3;`	Füllen des Arrays, so dass es danach den Inhalt {5, 0, 3} hat.
`a = new int[] {1, 1};`	Erzeugung eines neuen int-Arrays der Grösse 2. Initialisierung mit Inhalt {1, 1}. Zuweisung des Arrays zur Variablen `a`.
`int[] b = {1, 2, 1};`	Deklaration, Erzeugung und Füllen des Arrays in einem Schritt
`// a = {1, 2, 1};`	**Achtung:** Funktioniert **nicht** und ist deshalb auskommentiert

Speicher

Studiere den Code von `ArrayTests.memory()`. Führe ihn danach aus. Hast du eine Erklärung für das Phänomen?

Antwort: Lies den folgenden Ausschnitt aus Hanspeter Mössenböck, Sprechen Sie Java?

Arrayzuweisung

Einer Arrayvariablen dürfen alle Arrays zugewiesen werden, die vom passenden Elementtyp sind. Dem Array

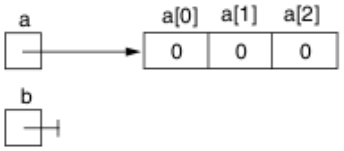
```
int[] a;
```

dürfen also beliebige int-Arrays zugewiesen werden, aber keine float-Arrays. Bei der Zuweisung wird jedoch nicht der *Wert* des Arrays in a gespeichert, sondern nur seine *Adresse*. Mehrere Arrayvariablen können daher auf dasselbe Array zeigen. Das ist für Programmieranfänger oft schwer zu verstehen, weshalb wir uns die Erzeugung und Zuweisung von Arrays nochmals anhand eines Beispiels anschauen. Im Codestück

```
int[] a, b;
a = new int[3];
```

werden zwei Arrayvariablen a und b deklariert, aber nur a zeigt auf ein Array, b ist noch leer. Die Variable b enthält den Wert null. null ist ein vordefinierter Zeigerwert, der so viel bedeutet wie »zeigt nirgendwo hin«. Er darf nicht mit dem int-Wert 0 verwechselt werden.

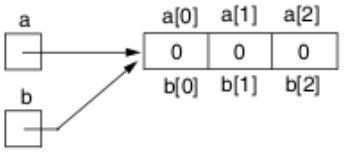
Elemente eines neu erzeugten Arrays werden in Java automatisch initialisiert. Numerische Elemente bekommen den Wert 0, boolean-Elemente den Wert false. Damit ergibt sich folgendes Bild (das Zeichen ⊣ bedeutet den Wert null):



Wir können nun die Zuweisung

```
b = a;
```

durchführen, was erlaubt ist, weil a auf ein int-Array zeigt und b ebenfalls int-Arrays referenzieren kann. Damit ergibt sich folgendes Bild:

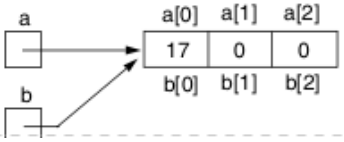


a und b zeigen jetzt auf das gleiche Array. Die Zuweisung ist also eine Zeigerzuweisung. Nur die in a gespeicherte Adresse des Arrays wird b zugewiesen, nicht das Array selbst. Arrayzuweisungen sind in Java immer Zeigerzuweisungen!

Die Elemente des Arrays können jetzt nicht nur als a[0] oder a[2] angesprochen werden, sondern auch als b[0] oder b[2]. Weist man a[0] einen neuen Wert zu, so ändert sich auch der Wert von b[0]. Nach der Zuweisung

```
a[0] = 17;
```

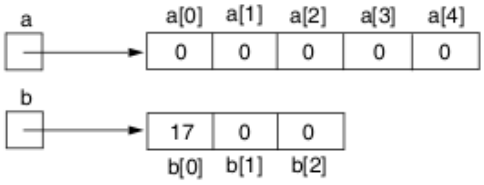
sieht das Array so aus:



Wenn man a nun ein neues Array zuweist, z.B.

```
a = new int[5];
```

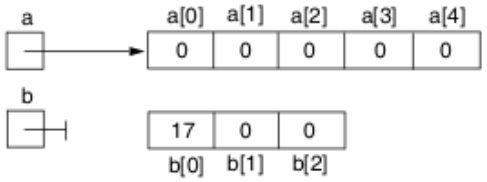
so ergibt sich folgendes Bild:



a und b zeigen jetzt wieder auf verschiedene Arrays. Interessant ist auch die Zuweisung

```
b = null;
```

Der Wert null kann jeder Arrayvariablen zugewiesen werden, wenn man will, dass die Variable auf kein Array zeigt. Damit ergibt sich folgendes Bild:

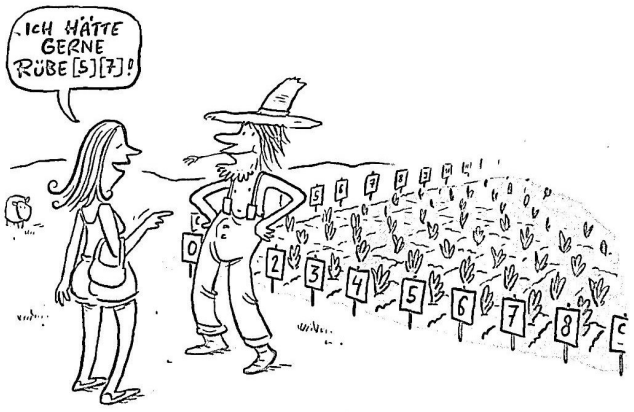


Das Array, auf das b zeigte, hängt nun »in der Luft«. Es wird von keiner Arrayvariablen mehr referenziert, und man kann es nicht mehr ansprechen. Mehr noch: Es gibt in diesem Beispiel keine Möglichkeit, je wieder einen Zeiger auf dieses Array verweisen zu lassen. Also wird es nicht mehr gebraucht und kann weggeworfen werden. Wenn ein Array nicht mehr referenziert wird, sorgt das Java-System automatisch dafür, dass sein Speicherplatz wieder freigegeben wird und für weitere Speicheranforderungen zur Verfügung steht.

Den Vorgang, dass unreferenzierte oder nicht mehr gebrauchte Elemente (z.B. auch ein Objekt/ eine Variable am Ende einer Prozedur oder Schleife) gelöscht werden, um Speicherplatz freizugeben, nennt man **Garbage Collection**. In Java passiert diese (im Gegensatz zu anderen Programmiersprachen) automatisch.

Übrigens: Das Verhalten, dass mit `=` und `==` auf den **Pointer (Zeiger)** Bezug genommen wird statt auf den Inhalt, zeigen auch Objekte (s. 2.1). Deshalb gibt es dafür die `equals()`-Methoden (s. später).

Zweidimensionale Arrays



Bildquelle: Christian Ullenboom, Java ist auch eine Insel, 16. Auflage

„Zweidimensionale Arrays“ entstehen, indem wir Arrays von Arrays erstellen, also zum Beispiel `int[][]`. So können zweidimensionale Situationen (Matrizen, Spielfelder für Spiele wie Schach, Tetris, Schiffe versenken etc.) einfach festgehalten werden.

Führe `TicTacToe.java` aus und stelle sicher, dass du alles verstanden hast.

Beachte: Wenn wir die Koordinaten als `[x][y]` festlegen, ist `x` der Index des Unterrarrays und `y` der Index des Elementes darin. D.h. unser Array ist ein Array von Spalten-Arrays, was (auf den ersten Blick unerwartet) eine „gespiegelte“ Ausgabe ergibt.

Aufgaben

Aufgabe 11. Löse `Aufgabe11.java`.

Aufgabe 12. Code lesen.

```java
import java.util.Arrays;

public class ArrayTests {

  static void test1() {
    double[] a;
    short[] b = new short[3];
    byte[] f = {0, 1};

    String[] c;
    c = {"saith", "wyth", "naw"};
    String[] d = c;
    c = new int[2];
    String[] e = new String[b[0]];
    String[] e = new String[] {d};
  }

  static void test2() {
    String[] a = { "un", "dau", "tri", "pedwar", "pump", "chwech" };
    String[] b = a;
    a[1] = "one";
    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
    b = new String[6];
    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
  }

  static int[] test3(int n) {
    int[] output = new int[n - 1];
    for (int i = 0; i < n; i++) {
      output[i] = i;
    }
    return output;
  }

  static int binSearch(int[] a, int value) {
    int low = 0;
    int high = a.length - 1;
    while (low <= high) {
      int mid = (low + high) / 2;
      if (a[mid] == value) {
        return mid;
      } else if (a[mid] < value) {
        low = mid + 1;
      } else { // a[mid] > value
        high = mid - 1;
      }
    }
    return -1;
  }

  public static void main(String[] args) {
    // 1) Welche Zeilen sind gültig, welche geben eine Fehlermeldung und warum?
    test1();

```

```
57      // 2) Was wird hier ausgegeben?
58      test2();
59
60      // 3) Was wird hier ausgegeben?
61      Arrays.toString(test3(2));
62
63      int[] orderedList = new int[] {1, 3, 4, 8};
64      // 4a) welche Variablen sind beim ersten Erreichen der Zeile 41 bzw. 43
        definiert, und welchen Wert haben sie?
65      System.out.println(binSearch(orderedList, 4));
66      // 4b) dito?
67      System.out.println(binSearch(orderedList, 2));
68    }
69  }
```

1.8 Rekursion

Methoden können sich selbst aufrufen. Dies wird **Rekursion (*recursion*)** genannt. Dabei wird auf dem Call-Stack eine neue Instanz der Methode erzeugt, die ihre eigenen Variablen enthält.
Rekursion ermöglicht manchmal elegante Lösungen für Programmier- und Alltagsprobleme:

```
static void threeWishes() {
  wish();
  wish();
  threeWishes();
}
```

Aufgabe 13. Nimm das Projekt `04_Rekursion` auf Github an und betrachte die Klasse `Recursion.java`. Studiere die Methode `factorial()`. Stelle bildlich dar, was im Call-Stack und in der Stack-Memory passiert, um zu verstehen, was darin passiert.

Studiere danach auch die Methode `binarySearch()` und überlege, wie sie funktioniert.

Aufgabe 14. Welche Ausgabe erfolgt beim Aufruf von `p(4)` bzw. `q(5)`?

```
static void p(int x) {
  if (x > 0) {
    p(x - 1);
  }
  System.out.println(x);
}

static int q(int x) {
  if (x == 0) {
    return 1;
  } else {
    int y = q(x - 1);
    System.out.println(x + y);
    return y + 1;
  }
}
```

Aufgabe 15. Löse `Aufgabe15.java`

2 Objektorientierte Programmierung

2.1 Klassen und Objekte

Objekte (*objects*) werden mit dem Schlüsselwort `new` erzeugt. Dabei müssen wir angeben, welcher Art (**Klasse (*class*), Typ (*type*)**) das Objekt ist, z.B. `new String` oder `new java.awt.Point` (vordefinierte Klasse aus der Java-Klassenbibliothek). Wir sagen auch, das Objekt sei eine **Instanz (*instance*)** der Klasse `String`/ `java.awt.Point`.

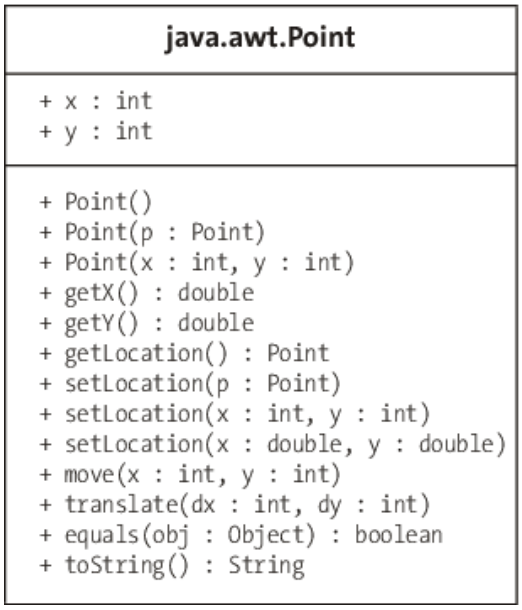


Abbildung 1: UML Class Diagram für `java.awt.Point`

Die Klasse bestimmt, was das Objekt für Eigenschaften und Methoden hat (ein `String`-Objekt hat zum Beispiel die Methode `length()`, die seine Länge zurückgibt). Diese werden in UML (*Unified Markup Language*)-Diagrammen dargestellt. In Abbildung 1 siehst du das UML-Diagramm für `java.awt.Point`. Die Klasse hat:

- Die **Eigenschaften/ Attribute (*attributes*)/ Felder (*fields*)** `x` und `y`, die für seine Position im Koordinatensystem stehen (komischerweise nur Ganzzahlen).
- **Konstruktor**-Methoden (**Konstruktoren/ *constructors***):

`Point p = new Point(2, 5);`	erzeugt einen neuen Punkt mit Koordinaten $(2, 5)$.
`Point q = new Point();`	erzeugt einen neuen Punkt mit Koordinaten $(0, 0)$.
`Point r = new Point(p);`	erzeugt eine Kopie eines Punktes `p` (d.h. im Speicher wird tatsächlich ein zweites Objekt angelegt und nicht nur ein neuer Zeiger auf das gleiche Objekt).

- **Objektmethoden (*instance methods*)**, d.h. Methoden, die mit der Syntax `p.methode()` auf einem Objekt `p` vom Typ `Point` aufgerufen werden:
  - **Getter- und Setter-Methoden**:

`p.setLocation(3, 4);`	macht das Gleiche wie `p.x = 3; p.y = 4;`
`p.getX()`	gibt die x-Koordinate von `p` zurück (seltsamerweise als `double` statt als `int`).

Getter- und Setter-Methoden existieren, da in den meisten Klassen (`java.awt.Point` ist hier eine Ausnahme) die Felder auf `private` gesetzt sind und deshalb von ausserhalb der Klasse nicht direkt gelesen oder verändert werden können. Dieses **Data Hiding** ist Teil des Prinzips der **Encapsulation**: Sie erlaubt, unerwartetes Verhalten zu vermeiden, indem z.B. Setter-Methoden Kontrollen ausführen, bevor sie den Feldern gewisse Werte zuweisen (s. Abschnitt 2.2). Der andere Vorteil der Encapsulation ist das Verbergen der inneren Struktur/ Mechanismen vor den Anwendern der Klasse, was erlaubt, die Implementation im Innern der Klasse zu ändern, solange das äussere Verhalten gleich bleibt.[2]

- Eine **`toString()`-Methode**, die das Objekt in einer sinnvoll lesbaren Form in einen `String` verwandelt (statt einen Hashcode/ Identifier[3] auszugeben). Das sieht dann je nach Java-Implementation etwa so aus: `java.awt.Point[x=10,y=9]`.
- Eine `equals()`-Methode, die feststellt, ob der Punkt gleich einem anderen ist. Dabei wird nicht wie mit `==` Objektgleichheit geprüft, sondern Inhaltsgleichheit, d.h. ob die beiden Punkte die gleichen Feldinhalte (also die gleichen Koordinaten) haben.
- Klassenspezifische Methoden:
  - `move()` und `translate()`, die das Erwartbare tun.
  - `p.getLocation()` macht das Gleiche wie `new Point(p)`.
    `q.setLocation(p);` macht das Gleiche wie `q.x = p.x; q.y = p.y;` (also eine Übernahme der Koordinaten, nicht der Zeiger).

    Die Methoden existieren hauptsächlich aus Kompatibilitätsgründen: Bei anderen geometrischen Objekten (z.B. `java.awt.Rectangle`) sind das Objekt und die „Location“ zwei unterschiedliche Dinge.

- Die Klasse hat keine **statischen/ Klassenmethoden (*static/ class methods*)**, d.h. Methoden, die unabhängig von einem konkreten `Point`-Objekt sind und deshalb mit der Syntax `Klasse.methode()` aufgerufen werden. Beispiele dafür aus anderen Klassen wären z.B. `Math.random()` (da muss nicht zuerst ein `Math`-Objekt erstellt werden), `Arrays.toString()` etc. Solche wären im UML-Diagramm unterstrichen dargestellt.

Aufgabe 1. Schreibe eine Klasse `PointTester.java`. Importiere die Klasse `java.awt.Point` vor dem Klassencode. Dann:

1. Erzeuge ein `Point`-Objekt mit Koordinaten $(2, 5)$, speichere es in einer Variablen und gib seinen Inhalt auf der Konsole aus (indem du implizit oder explizit `toString()` aufrufst).
2. Vertausche die beiden Koordinaten des Punktes. Greife dabei nicht direkt auf seine Felder zu, sondern brauche die Getter- und Setter-Methoden.
3. Erzeuge mit möglichst wenig Code einen zweiten Punkt, der die Koordinaten des ersten übernimmt. Teste die beiden Punkte auf Objekt- und Inhaltsgleichheit.

[2] Zu Encapsulation und den anderen drei Prinzipien der objektorientierten Programmierung siehe https://stackify.com/oops-concepts-in-java/

[3] Hashcode = effizient, aber nicht rückverfolgbar berechneter Ausdruck, der möglichst einzigartig (und damit eindeutig) für das betreffende Objekt ist. Hashcodes werden gebraucht, um Objekte quasi-eindeutig zu identifizieren, deshalb der Begriff *identifier*.

2.2 `Fraction` - Eigene Klassen schreiben, Encapsulation

Wir wollen eine Klasse `Fraction` schreiben, mit der wir rechnen können. Dazu habe ich bereits eine Datei `Fraction.java` mit den Feldern `numerator` und `denominator` erstellt. Ebenso existiert eine Datei `FractionTester.java` für Testcode.

1. Probiere den Testcode in `FractionTester` aus. Es wird ein konkretes `Fraction`-Objekt `f` nach der „Vorlage“ der Klasse `Fraction` erstellt. Stelle sicher, dass du alles verstehst.

2. **`toString()`:** Die Zeile mit dem `println()`-Befehl ist etwas umständlich, aber leider liefert `System.out.println(f)` ja nur einen Hashcode. Das wollen wir ändern!

   Füge der Klasse `Fraction` eine Funktion `toString()` hinzu. Am Einfachsten geht das mit *Rechtsklick/ Source Action/ Generate toString()*. Passe den Code der automatisch generierten `toString()`-Methode an, um eine sinnvolle Bruchausgabe zu bekommen.

   `@Override` heisst übrigens, dass die `toString()`-Methode der Klasse `Object` (die einfach nur den Hashcode generiert und ausgibt) überschrieben wird.

3. **Encapsulation, Getter und Setter:** Zeile 6 in `FractionTester` macht mathematisch natürlich keinen Sinn. Genau um solchen Quatsch abfangen zu können, soll der Wert `denominator` nicht von aussen veränderbar sein, sondern nur von Prozeduren innerhalb der Klasse (die aber ihrerseits `public` sind, also von aussen aufgerufen werden können). Setze deshalb das Schlüsselwort `private` vor `int numerator, denominator;`. Nun funktionieren die Zeilen 5-7 im Testcode nicht mehr. Probiere es aus und studiere die Fehlermeldung!

   Wir brauchen nun also Methoden, um auf `numerator` und `denominator` zuzugreifen. Eine davon, `setDenominator()`, existiert bereits. Sie produziert einen Fehler, wenn der Nenner auf 0 gesetzt wird. Probiere es in der Tester-Klasse aus! Studiere danach den Code. Das Schlüsselwort `this` ist ein Platzhalter für das Objekt, auf dem die Methode aufgerufen wird. Dadurch kann das Methodenargument `denominator` vom Feld `this.denominator` des `Fraction`-Objekts unterschieden werden.

   Erstelle nun die fehlenden Getter- und Setter-Methoden. Am Einfachsten geht das per *Rechtsklick/ Source Action/ Generate Getters* bzw. *Setters*. Studiere den generierten Code kurz. Weise danach dem `Fraction`-Objekt `f` im Testcode gültige Werte für Zähler und Nenner zu.

4. **Konstruktoren:** Das Erzeugen von Brüchen mit einzelner Zuweisung der Werte von `numerator` und `denominator` ist mühsam. Wenn wir eine sogenannte Konstruktor-Methode schreiben, können wir den Bruch in einem Schritt erzeugen und initialisieren. Eigentlich sollten Konstruktor-Methoden Initialisierungs-Methoden heissen, denn sie initialisieren lediglich die Felder des soeben erzeugten Objekts. Konstruktor-Methoden werden normalerweise am Anfang der Klasse positioniert, gleich nach der Definition der Felder. Erzeuge einen Konstruktor mit *Rechtsklick/ Generate Constructors*, hake beide Felder an und drücke OK. Studiere den Code.

   Hier sehen wir wieder die Zuordnung mit `this`. Ersetze die Zeile `this.denominator;` durch `this.setDenominator(denominator);`. Indem wir den Setter verwenden, haben wir gleich die Wertprüfung für ungültige Nenner wieder dabei.

   Wenn du jetzt `FractionTester` ausführst, bekommst du einen Fehler. Dadurch, dass wir

einen neuen Konstruktor geschrieben haben, wird der argumentlose Standard-Konstruktor `Fraction()` ungültig. Das ist aber eine gute Sache. Mit `new Fraction()` wurde nämlich ein `Fraction`-Objekt mit nicht-initialisierten Feldern erzeugt. Nicht-initialisierte `int`-Variablen haben bekanntlich standardmässig den Wert 0, d.h. unser Bruch `f` hatte den Wert $\frac{0}{0}$, bevor wir ihm Werte zugewiesen haben…

Erzeuge in der Testklasse ein neues `Fraction`-Objekt, das den neuen Konstruktor braucht.

Erzeuge ausserdem einen argumentlosen Konstruktor `Fraction()` mit einer einzigen Zeile `this(0, 1);`. Die Methode `this(int, int)` steht für den Aufruf der bereits erstellten Konstruktormethode `Fraction(int, int)`. Dieser Aufruf muss in Konstruktoren immer an erster Stelle stehen. Die Initialisierung von `Fraction`-Objekten mit $\frac{0}{1}$ macht mehr Sinn als mit $\frac{0}{0}$. Teste wieder.

5. **Copy-Konstruktor und `equals()`:** Die Zuweisung `g = f;` ist für zwei `Fraction`-Variablen bekanntlich lediglich eine Zeigerzuweisung, und im Speicher existiert nur ein `Fraction`-Objekt. Um `Fraction`-Objekte echt zu kopieren, brauchen wir einen Copy-Konstruktor. Entkommentiere dazu den Code `public Fraction(Fraction f){...}`).

   Der entkommentierte Konstruktor erstellt ein neues `Fraction`-Objekt und übergibt ihm die Werte der übergebenen Fraction `f`. Probiere es aus, indem du damit in der Testklasse eine Kopie `g` von `f` erstellst. Überprüfe wieder Objekt- und Inhaltsgleichheit.

   Inhaltsgleichheit konntest du nur „manuell“ im Debugging-Modus oder durch Vergleich der `toString()`-Outputs überprüfen. Um das zu ändern, können wir eine `equals()`-Methode schreiben. Auch diese kannst du einfach entkommentieren. Studiere sie. Warum macht der Code (mathematisch) Sinn?

   Teste die Methode danach an `f` und `g`, aber auch mit einem weiteren Bruch, der den gleichen Wert, aber unterschiedlichen Zähler und Nenner hat wie einer der beiden.

6. **Statische und dynamische Methoden (Klassen- und Instanzenmethoden):** Jetzt fehlen noch Operationen wie Addition, Subtraktion etc. Die Addition ist bereits geschrieben und kann entkommentiert werden. Du siehst, dass zwei Methoden `add()` existieren:

   - Die **Klassenmethode (=statische Methode)**, die mit `Fraction.add(f, g)` aufgerufen wird und das Resultat als (neues) `Fraction`-Objekt zurückgibt. Die Methode wird mit dem Schlüsselwort `static` zu einer Klassenmethode gemacht.
   - Die **Objektmethode (=dynamische Methode)**, die mit `f.add(g)` aufgerufen wird und die `g` zu `f` dazuaddiert (und das Resultat direkt in `f` speichert). Sie hat keine Rückgabe.

   Je nach Kontext macht die eine oder andere Methode mehr Sinn. Die Klassenmethode kommt zum Zug, wenn bestehende Objekte nicht verändert werden sollen. Die Objektmethode hingegen kann sinnvoll sein, wenn (Speicher-)Effizienz gefragt ist.

   Teste beide Methoden.

7. **Statische Variablen:** Es gibt neben Methoden auch die Möglicheit, Variablen als statisch zu definieren. Die bereits bestehende Variable `numberOfFractions` kann z.B. gebraucht werden, um zu zählen, wie viele `Fraction`-Objekte bereits erzeugt wurden. Im Konstruktor müssen wir dann die Zeile `numberOfFractions++;` hinzufügen (nur im ersten Konstruktor, da die anderen ja darauf zugreifen). Probiere es aus!

8. **Konstanten:** Der Vollständigkeit halber und um Konfusion mit `static` zu vermeiden: Das Schlüsselwort `final` kennzeichnet Konstanten, also Variablen mit unveränderlichem Wert. Auch Methoden können `final` sein, was heisst, dass sie in abgeleiteten Klassen nicht überschrieben werden können. `final` und `static` können unabhängig voneinander gesetzt werden.

   In diesem Kontext noch etwas zu **Namenskonventionen**:

   - **Schlüsselwörter** werden klein geschrieben: `public`, `static`, `for`, `if`
   - **primitive Datentypen** ebenso: `int`, `float`, `char`
   - **Klassen**namen sind Substantive (da sie Vorlagen für Objekte sind) und beginnen mit einem Grossbuchstaben: `Fraction`, `String`, `Tester`, `FractionTester`
   - **Methoden** tun etwas und haben deshalb Verbnamen (Camel Case: klein beginnen, danach jedes Wort gross): `getNumerator()`, `add()`, `move()`, `killAllOpenWindows()`
   - **Variablen**namen sind auch in Camel Case: `variable`, `myVariable`, `numberOfElements`
   - **Konstanten** werden durchgehend gross geschrieben und verwenden (als einzige) Unterstriche als Worttrenner: `PI`, `EULERS_NUMBER`, `EARTH_CIRCUMFERENCE`

   Aufgabe 2.

   a) Definiere in der Testklasse mit `final double PI = 3.14159;` eine Konstante und versuche ihren Wert zu ändern.

   b) Erkläre den gesamten Code `public static void main(String[] args)`.

9. **Sichtbarkeit:** Die Sichtbarkeit von Variablen und Methoden kann mit den Schlüsselwörtern (***access modifiers***) `public` (+), `protected` (#) und `private` (-) beeinflusst werden:

sichtbar…	`public`	`protected`	(ohne Modifier)	`private`
…in der Klasse selbst	ja	ja	ja	ja
…in Klassen des gleichen Pakets	ja	ja	ja	-
…in abgeleiteten Klassen	ja	ja	-	-
…global	ja	-	-	-

   Bemerkungen:

   - +, #, ∼ (= ohne Modifier) und - sind die Notationen, die in UML-Diagrammen gebraucht werden (s. Abb. 1).
   - **Pakete** fassen mehrere Klassen zusammen, die inhaltlich oder funktionell zusammengehören. So ist z.B. `java.awt` ein Paket, das Klassen für User Interfaces und graphische Darstellungen bereitstellt (eben z.B. `Point.java`).
   - **Abgeleitete Klassen (*subclasses*)** sind Erweiterungen von Klassen, die alle Eigenschaften und Methoden der ursprünglichen Klasse erben (sie können aber überschrieben werden). So ist etwa jede Klasse (implizit) von `Object` abgeleitet und erbt Methoden wie z.B. `toString()` oder den argumentlosen Konstruktor. Wenn wir eine Klasse `Square` schreiben würden, könnte sie von einer Klasse `Rectangle` abgeleitet sein, die wiederum von einer Klasse `Quadrilateral` abgeleitet ist.

   Aufgabe 3. Erstelle ein UML-Diagramm für die Klasse `Fraction`. Stelle dabei Sichtbarkeit dar und ob eine Variable/ Methode statisch ist.

Aufgabe 4. Ergänze die Klasse `Fraction` mit weiteren sinnvollen Methoden. Oft ist nicht nur das Programmieren selbst, sondern bereits das sorgfältige Planen der Implementation eine wertvolle Übung. Möglichkeiten sind:

1. Implementiere `multiply()` und danach auch `subtract()` und `divide()` sowohl als Klassen- als auch als Instanzenmethode. Schreibe so wenig wie möglich doppelten Code.
2. Implementiere Encapsulation/ Data Hiding für die Variable `numberOfFractions`. Was brauchst du dazu alles?
3. Eine Option ist, dass Brüche immer vollständig gekürzt abgespeichert werden. Schreibe dazu eine Methode `simplify()` (= der korrekte englische Fachbegriff für „kürzen“!). Eine bereits programmierte Methode aus einem früheren Kapitel wird hier nützlich sein. Überlege, wo im Code du `simplify()` überall aufrufen musst. Soll die Methode `public` sein? Überlege auch, wie du mit Vorzeichen umgehst.
4. Schreibe zusätzliche Methoden `add()`, `multiply()` etc., die statt zwei Brüchen einen Bruch und eine Ganzzahl miteinander verrechnen (Method overloading).
5. Verändere `toString()`, im Fall eines Nenners 1 eine ganze Zahl ausgegeben wird und im Fall von Werten betragsmässig grösser als 1 eine gemischte Zahl.
6. Schreibe eine abgeleitete Klasse `MixedNumber` (beginnend mit `public class MixedNumber extends Fraction`). Welche Felder musst du zusätzlich einführen, welche Methoden überschreiben/ ergänzen? Mit `super.` kannst du auf Felder und Methoden der Mutterklasse `Fraction` zugreifen.