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.