Einführung in die Programmierung mit Java

Dozent:	Prof. Dr. Michael Eichberg
Kontakt:	michael.eichberg@dhbw.de, Raum 149B

Version:	1.3
Quelle:	Teilweise inspiriert von oder basierend auf Lehrmaterial von Prof. Dr. M. Matt bzw. Prof. C. Binning.

Folien:	https://delors.github.io/prog-java-basics/folien.de.rst.html https://delors.github.io/prog-java-basics/folien.de.rst.html.pdf
Kontrollfragen:	https://delors.github.io/prog-java-basics/kontrollfragen.de.rst.html
Übungsaufgaben:	klausurvorbereitung.de.rst.html
Fehler melden:	https://github.com/Delors/delors.github.io/issues

1. Einführung

2

"Hello World" - das erste Java-Programm

`HelloWorld.java`[1]

```
1 void main() {
2     IO.println("Hello World!");
3 }
```

Die Ausführung des Programms ist (in der Konsole/im Terminal) mit `java HelloWorld.java` möglich.

Die Datei enthält ein einfaches Java-Programm, das den Text `Hello World!` auf der Konsole ausgibt.

In der ersten Zeile wird die Methode `main` definiert. Diese ist die Einstiegsmethode in das Programm. Der Text `Hello World!` wird mit der *Methode* `println` auf der Konsole ausgegeben. `IO.print(...)`, und `IO.println(...)` sind in Java-Skripten immer verfügbar und geben den übergebenen Text auf der Konsole aus. Die Methode `print` tut dies ohne und die Methode `println` mit Zeilenumbruch (`\n`) am Ende.

---

[1] Die Datei `HelloWorld.java` kann hier heruntergeladen werden. Java 25 ist notwendig!

Von der Konsole lesen

`HelloYou.java`[2]

```
1 void main() {
2     IO.println("Hello " + IO.readln("What is your name? "));
3 }
```

Mit Hilfe von `readln` können Sie von der Konsole lesen. In Java-Skripten ist `readln` immer verfügbar. Das Programm gibt den Text `Hello` gefolgt von dem eingegebenen Text aus. Die Methode `readln` gibt erst den übergebenen String aus und liest dann eine Zeile von der Konsole ein. Der eingelesene Text wird dann an das Wort "Hello " angehängt (mittels des "+" Operators) und als ganzes zurückgegeben.

---

[2] HelloYou.java

Ausführung von Java-Skripten

- Java-Skripte können direkt mit `java <Datei mit Sourcecode>` ausgeführt werden.

  Ggf. ist die Option `--enable-preview` notwendig; diese ermöglicht die Verwendung der neuesten, noch in der Entwicklung befindlichen, Features von Java.

- Java-Skripte können auf Unixoiden Betriebssystemen (insbesondere Linux, MacOS, BSD) auch direkt ausgeführt werden, wenn sie ein Shebang (`#!`) in der ersten Zeile enthalten, die Datei *nicht* mit `.java` endet und die Dateirechte (ausführbar) entsprechend gesetzt sind (Auf Unixoiden: `-rwxr-xr-x` … HelloWorldScript):

```
1 #!/usr/bin/env java --source 25 -ea
2 // RECALL: THE FILENAME MUST NOT END WITH ".java" ACCORDING TO JEP 330!
3
4 void main() {
5     IO.println("Hello World!");
6 }
```

Übung - mein erstes Programm

1.1. Lesen von und Schreiben auf die Konsole

Schreiben Sie ein Java-Programm (`GutenMorgen.java`), das erst nach dem Namen des Nutzers `X` fragt und dann `Guten Morgen X!` auf der Konsole ausgibt. Beachten Sie dabei, dass der Text `X` durch den eingegebenen Namen ersetzt wird und am Ende ein Ausrufezeichen steht.

Als zweites soll das selbe Programm dann nach dem Wohnort `Y` des Nutzers fragen und dann `Y ist wirklich schön!` auf der Konsole ausgeben.

Schreiben Sie das Programm und führen Sie es aus!

※ Hinweis

Vorgehensweise:

1. Stellen Sie sicher, dass Java korrekt installiert ist. Öffnen Sie dazu die Konsole und geben Sie java --version ein.
2. Öffnen Sie einen Texteditor (z. B. Visual Studio Code oder ZED oder ...)
3. Schreiben Sie den Rumpf des Programms: `void main() { <IHR CODE> }`
4. Ersetzen Sie `<IHR CODE>` durch den Code, der den Nutzer nach seinem Namen X fragt und dann "Guten Morgen X!" ausgibt.
5. Führen Sie den Code aus in dem Sie die Konsole/ein Terminal öffnen und dort: `java --enable-preview GutenMorgen.java` ausführen.

Übung

1.2. Java-Skripte

Zur Ausführung der vorhergehenden Programme mussten Sie immer den Java Interpreter starten und das Programm ausführen. Stellen Sie Ihr letztes Programm so um, dass eine direkte Ausführung - wenn Java korrekt installiert ist - möglich ist.

Beispielausführung auf der Kommandozeile:

```
$ ./GutenMorgen
Wie ist Dein Name? Michael
Hallo Michael!
Wo wohnst Du? In einem schönen Ort
In einem schönen Ort ist wirklich schön!
```

2. Einfache Prozedurale Programmierung mit Variablen, Konstanten, Literalen und Ausdrücken

Prozedurale Elemente

Kommentare: Dienen der Codedokumentation und werden vom Compiler ignoriert.

primitive Datentypen: Ganze Zahlen (`byte`, `int`, `long`), Fließkommazahlen (`float`, `double`), Zeichen (`char`, `byte`) und Wahrheitswerte (`boolean`).

Literale: Konstante Wert (z.B. `42`, `3.14`, `'A'`, `"Hello World!"`, `true`).

Zuweisungen: Speichern eines Wertes in einer Variable.
(*Eine benannten Stelle im Speicher*)

Ausdrücke: Dienen der Berechnung von Werten mit Hilfe von Variablen, Literalen und Operatoren.

Kontrollstrukturen: Dienen der Ablaufsteuerung mit Hilfe von Schleifen (`while`, `do-while`, `for`) und Verzweigungen (`if-else`, `switch-case`).

Unterprogramme: Methoden (*Prozeduren* und *Funktionen*), die eine bestimmte Funktionalität wiederverwendbar bereitstellen.

Kommentare

- Kommentare dienen der Dokumentation des Codes und helfen anderen Entwicklern den Code zu verstehen.
- In Java unterscheiden wir folgende Arten von Kommentaren:
  - Einzeilige Kommentare, die mit `// beginnen und bis zum Ende der Zeile gehen.`
  - Mehrzeilige Kommentare, die mit `/* beginnen und mit */` enden.

    Kommentare, die mit `/**` beginnen und mit `*/` enden, sind so genannte JavaDoc Kommentare und dienen der Erzeugung von Dokumentation.
  - [ab Java 23] Mehrzeilige Kommentare, bei der jede Zeile mit `///` beginnt, werden als Markdown basierte JavaDoc Kommentare interpretiert.

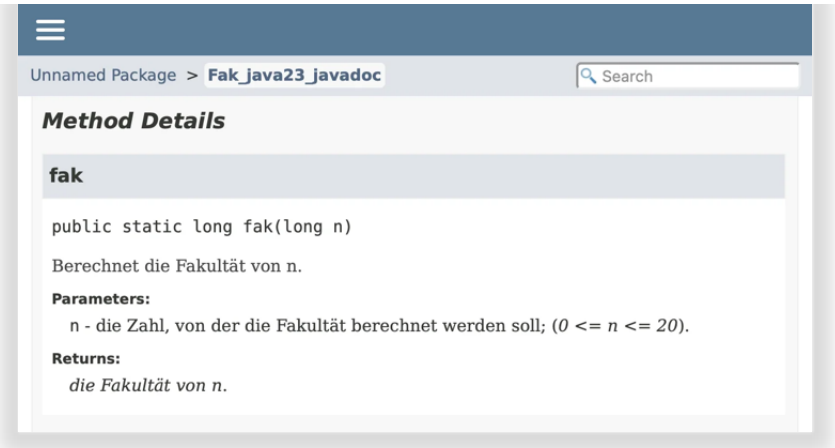
Beispiel (ab Java 1.0 - spezifische Tags und HTML)

```java
/**
 * Berechnet die Fakultät von n.
 *
 * @param n die Zahl, von der die Fakultät berechnet werden soll; (0 ⩽ n ⩽ 20).
 * @return die Fakultät von n.
 */
long fak(long n){ // TODO mögliche Fehlerfälle abfangen
    /* Die Verwendung von long als Datentyp limitiert uns auf n ⩽ 20;
       durch den wechsel von long auf double könnten wir bis n ⩽ 170 rechnen;
       sind aber unpräziser. */
    if (n == 0) return 1;
    else return n * fak(n-1);
}
```

Beispiel (ab Java 23 - spezifische Tags und Markdown)

```java
/// Berechnet die Fakultät von n.
///
/// @param n die Zahl, von der die Fakultät berechnet werden soll;/
///          (*0 ⩽ n ⩽ 20*).
/// @return _die Fakultät von n_.
long fak(long n){ // TODO mögliche Fehlerfälle abfangen
    /* Die Verwendung von long als Datentyp limitiert uns auf n ⩽ 20;
       durch den wechsel von long auf double könnten wir bis n ⩽ 170 rechnen;
       sind aber unpräziser. */
    if (n == 0) return 1;
    else return n * fak(n-1);
}
```

Erzeugte Dokumentation (mit Java 23)



JavaDoc tags

@param <name descr>: Dokumentiert einen Parameter einer Methode.

@return <descr>: Dokumentiert den Rückgabewert einer Methode.

Java Shell

- Die Java Shell (`jshell`) ist ein interaktives Werkzeug, das es ermöglicht Java-Code (insbesondere kurze Snippets) direkt auszuführen.
- Starten Sie die Java Shell mit dem Befehl `jshell --enable-preview` in der Konsole.
- Den gültigen Java-Code können Sie direkt in der Java Shell eingeben oder über `/edit` als Ganzes bearbeiten.
- Sie beenden die Java Shell mit dem Befehl `/exit`.
- Die Java Shell eignet sich insbesondere für das Ausprobieren von Code-Schnipseln und das Testen von Methoden.

```
# jshell --enable-preview

|  Welcome to JShell -- Version 25
|  For an introduction type: /help intro

jshell> var x = "X";
x ==> "X"

jshell> x + "Y"
$2 ==> "XY"

jshell> $2.length()
$3 ==> 2
```

Übung - Java als Taschenrechner

2.1. Rechnen auf der Konsole

Verwenden Sie die JShell als Taschenrechner und lösen Sie die folgenden Aufgaben in der angegebenen Reihenfolge jeweils mit Hilfe von *einer* Formel:

1. Berechnen Sie, wie viele Sekunden ein Schaltjahr hat.
2. Sie nehmen einen Kredit über 47865 € auf und zahlen monatlich 3,6% Zinsen. Wie viele Zinsen haben Sie nach 5 Jahren bezahlt?
3. Ein Bauer hat 120 Äpfel. Er möchte die Äpfel gleichmäßig auf 4 Körbe verteilen. Nachdem er die Äpfel aufgeteilt hat, isst er 5 Äpfel aus jedem Korb. Wie viele Äpfel hat er noch?
4. Nehmen Sie an, dass weltweit jeden Tag 1 500 000 000 Plastikflaschen produziert werden. Wie viele Flaschen werden in einem Jahr produziert, wenn das Jahr 365 Tage hat, aber an den Wochenenden nicht produziert werden würde (gehen Sie von 52 Wochenenden aus)?

Zum Starten der JShell müssen Sie die Konsole (ein Terminal) öffnen und `jshell` eingeben.

※ Hinweis

- In Programmiersprachen wird generell die englische Schreibweise für Zahlen verwendet. D. h. Sie müssen das Dezimalkomma durch einen Punkt ersetzen.)
- Die Division wird in (den meisten) Programmiersprachen mit dem Operator `/` durchgeführt.
- Die Multiplikation wird in (den meisten) Programmiersprachen mit dem Operator `*` durchgeführt.
- Sie können Klammern (`(` und `)`) so verwenden, wie Sie es von der Mathematik gewohnt sind.
- Sie können große Zahlen mit einem Unterstrich (_) formatieren, um die Lesbarkeit zu erhöhen: z. B. `1_500_000_000`.

3. Primitive Datentypen

Arten und Verwendung von Datentypen

Um die erlaubten Werte von Parametern, Variablen und Rückgabewerten genauer spezifizieren zu können, werden Datentypen verwendet. Java stellt hierzu (unter anderem) primitive Datentypen zur Verfügung.

Ein primitiver Datentyp ist z. B. `int` (d. h. *integer* bzw. *Ganzzahl*).

Dieser Datentyp legt fest, dass ein Wert eine Ganzzahl mit dem Wertebereich: `[-2147483648, 2147483647]` ist.

Alle von Java unterstützten primitiven Datentypen

Art	Datentyp	Beispiel
Ganzzahlen	`byte`, `short`, `int`, `long`	`123`
Fließkommazahlen *Floating-Point Numbers*	`float`, `double`	`1.23` oder `3.141d`
Zeichen	`char`	`'a'`
Wahrheitswerte	`boolean`	`true`

Bitte beachten Sie, dass in Code für Zahlen immer die Englische Schreibweise verwendet wird. D. h. das Dezimalkomma wird durch einen Punkt ersetzt.

Java kennt neben den primitiven Datentypen auch noch Arrays, Aufzählungen (`enum`) sowie Klassen und Interfaces. Diese werden wir später behandeln.

Ganzzahlige Datentypen - Hintergrund

- Ganzzahlige Werte werden im Speicher als Binärzahlen gespeichert; d. h. als Folge von Nullen und Einsen.
- Um verschieden große Werte zu speichern, stellen Programmiersprachen ganzzahlige Werte mit einer unterschiedlichen Zahl von Bits dar.

Zahlen werden immer mit 8 Bit (1 Byte), 16 Bit (2 Byte), 32 (4 Byte) oder 64 Bit (8 Byte) gespeichert.

※ Hinweis

In Java werden Zahlen immer vorzeichenbehaftet gespeichert. D. h. ein Bit wird für das Vorzeichen verwendet; auch wenn es nicht immer benötigt wird.

Umrechnung Binär-Dezimal

Binär	Dezimal
`0000 0000`	+0
`0000 0001`	+1
...	...
`0111 1111`	+127
`1000 0000`	-128
...	...
`1111 1111`	-1

Datentyp	Genauigkeit (in Bit)	Wertebereich	Anzahl Werte
`byte`	8	-128 bis 127	2^8
`short`	16	-32768 bis 32767	2^{16}
`int`	32	-2147483648 bis 2147483647	2^{32}
`long`	64	-92233720368547758080 bis 92233720368547758079	2^{64}

Die Größenwahl für `long` und `int` ist teilweise historisch bedingt. Auf gängigen Prozessoren sind jedoch 64 Bit und 32 Bit die natürlichen Größen für Ganzzahlen und können effizient verarbeitet werden.

Gleitkommatypen - Hintergrund (Konzeptionell)

Gleitkommazahlen werden in Java nach Norm IEEE 754 (Seit Java 15 Version 2019) durch die Mantisse m und den Exponent e dargestellt: $z = m \times 2^e$.

Für das Vorzeichen wird das erste Bit verwendet, für Mantisse und Exponent werden zusammen 31- (bei `float`) bzw. 63-Bit (bei `double`) verwendet.

Die Mantisse und der Exponent sind vorzeichenbehaftete Ganzzahlen.

Beispiel (vereinfacht)

$7 \times 2^{-1} = \frac{7}{2} = 3.5$

$-7 \times 2^{-1} = \frac{-7}{2} = -3.5$

$7 \times 2^{-3} = \frac{7}{8} = 1.125$

$7 \times 2^{0} = \frac{7}{1} = 7$

Datentyp	Genauigkeit	Mantisse	Exponent	Wertebereich
`float`	32	23	8	ca. -3,4·10^{38} bis 3,4 · 10^{38}
`double`	64	52	11	ca. -1,8·10^{308} bis 1,8 · 10^{308}

Ganzzahlen $< 2^{24}$ können bei Verwendung des Datentyps `float` exakt dargestellt werden; bei `double` sind es Ganzzahlen $< 2^{53}$.

In beiden Fällen gibt es noch die Möglichkeit +/- Unendlich und NaN (Not a Number) zu repräsentieren.

Gleitkommatypen - Verwendung

Warnung

Bei Berechnungen mit Gleitkommazahlen treten Rundungsfehler auf, da nicht alle Werte in beliebiger Genauigkeit dargestellt werden können

Beispiel: Der Wert `0.123456789f` (`float`) wird durch die Darstellung mit Mantisse und Exponent ($m \times 2^e$) zu `0.12345679`.

Gleitkommazahlen sind somit nicht für betriebswirtschaftliche Anwendungen geeignet.

Gleitkommazahlen sind z. B. für wissenschaftliche Anwendungen geeignet.

Für betriebswirtschaftliche Anwendungen gibt es den Datentyp `BigDecimal`. Dieser ist aber kein primitiver Datentyp und wird später behandelt.

Zeichen - Hintergrund

- einzelne Zeichen (z. B. 'a') werden in Java mit dem Datentyp `char` dargestellt
- ein `char` ist (intern) eine vorzeichenlose Ganzzahl mit 16 Bit (d. h. eine Zahl im Bereich `[0,65536]`), die den Unicode-Wert des Zeichens repräsentiert

  Alle gängigen (westeuropäischen) Zeichen können mit einem `char` dargestellt werden.

  Warnung

  Seit Java eingeführt wurde, wurde der Unicode Standard mehrfach weiterentwickelt und heute gibt es Zeichen, die bis zu 32 Bit benötigen. Diese können mit nur einem `char` nicht dargestellt werden und benötigen ggf. zwei `char`s.
- Für Zeichenketten (z. B. `"Hello World"`) existiert ein nicht-primitiver Datentyp `String`.

Unicode Zeichen und `char`s

Hinweise: - 0x1F60E ist der Unicode Codepoint von 😎 und `Character.toChars(<Wert>)` rechnet den Wert um. - In Java ist die Länge (`<String>.length()`) einer Zeichenkette (📼 *String*) die Anzahl der benötigten `char`s und entspricht somit nicht notwendigerweise der Anzahl der (sichtbaren) Zeichen.

```
jshell> var smiley = Character.toChars(0x1F60E)
smiley ==> char[2] { '?', '?' }

jshell> var s = new String(smiley)
s ==> "😎"

jshell> s.length()
$1 ==> 2

jshell> s.getBytes(StandardCharsets.UTF_8)
$2 ==> byte[4] { -16, -97, -104, -114 }

jshell> s.codePointCount(0,s.length())
$3 ==> 1
```

Wahrheitswerte (Boolesche) - Hintergrund

- die Wahrheitswerte wahr (`true`) und falsch (`false`) werden in Java mit dem Datentyp `boolean` dargestellt
- häufigste (explizite) Verwendung ist das Speichern des Ergebnisses einer Bedingungsüberprüfung

  (Wahrheitswerte sind zentral für Bedingungsüberprüfungen und Schleifen, werden dort aber selten explizit gespeichert; z. B. beim Test von `n` auf 0 im Algorithmus für die Berechnung der Fakultät.)

Konvertierung von Datentypen

- Die (meist verlustfreie,) implizite Konvertierung von Datentypen ist nur in eine Richtung möglich:

  ((byte → short) | char) → int → long → float → double

- Konvertierungen in die andere Richtung sind immer explizit anzugeben, da es zu Informationsverlust kommen kann

  Beispiel: `int` zu `byte` (Wertebereich `[-128,127]`)

  Bei der Konvertierung von `int` zu `byte` werden die höherwertigen Bits (9 bis 32) einfach abgeschnitten.

  (byte) 128 ⇒ -128

  (byte) 255 ⇒ -1

  (byte) 256 ⇒ 0

- Beispiel für die verlustbehaftete implizite Konvertierung

```
jshell> long l = Long.MAX_VALUE - 1;
l ==> 9223372036854775806

jshell> float f = l
f ==> 9.223372E18

jshell> f == l
$1 ==> true                    // Warum ?

jshell> ((long) f) == l
$2 ==> false

jshell> ((long) f)
$3 ==> 9223372036854775807     // == Long.MAX_VALUE
```

- Wahrheitswerte können nicht konvertiert werden.

4. Literale

Literale - Übersicht

Literale stellen konstante Werte eines bestimmten Datentyps dar.

Datentyp	Literal in Java-Code (Beispiele)
`int`	Dezimal: 127 ; Hexadezimal: 0xcafebabe[3] ; Oktal: 010 ; Binär: 0b1010
`long`	123_456_789l oder 123456789L ("_" dient nur der besseren Lesbarkeit)
`float`	0.123456789f oder 0.123456789F
`double`	0.123456789 oder 0.123456789d oder 0.123456789D
`char`	'a' (Zeichen-Darstellung) oder 97 (Zahlen-Darstellung) oder '\u0061' (Unicode-Darstellung) oder Sonderzeichen (siehe nächste Folie)
`String`	"Hallo" oder """ Text-block"""
`boolean`	true oder false

Textblöcke werde seit Java 15 unterstützt.

Mittels: `-Xlint:text-blocks` können Sie sich warnen lassen, wenn die Textblöcke potentiell nicht korrekt formatiert sind.

---

[3] 0xcafebabe ist der Header aller kompilierten Java-Klassen-Dateien.

Literale - Sonderzeichen ("\" ist das Escape-Zeichen)

Datentyp	Literal (Beispiele)
\'	Einfaches Hochkomma
\"	Doppeltes Hochkomma
\ \	Backslash
\b	Rückschrittaste (backspace)
\f	Seitenvorschub (form feed)
\n	Zeilenschaltung (line feed)
\t	Tabulator
\r	Wagenrücklauf

5. Variablen und Konstanten

Variablen - Übersicht

- Variablen stellen einen logischen Bezeichner für einen Wert eines bestimmten Datentyps dar.
- Variablen müssen erst deklariert werden. Danach können sie weiter initialisiert werden, wenn der Standardwert nicht ausreicht.

  Deklaration:

  Variablennamen und Datentyp werden festgelegt.

  Initialisierung (optional):

  Variablen werden mit einem bestimmten Wert versehen.
- der Wert einer Variablen kann jederzeit geändert werden

Beispieldeklaration und -initialisierung

```
1  void main() {
2      // Deklaration (Datentyp muss konkret angegeben werden)
3      int alter;
4      // Deklaration und Initialisierung inkl. Datentyp (Standarfall)
5      String name = "Asta Mueller";
6
7      // vereinfachte Deklaration mittels var und Initialisierung (seit Java 10)
8      // (Datentyp wird automatisch erkannt)
9      var geburtsOrt = "Berlin";
10     var wohnort = "Schönau";
11     var geschlecht = 'd';
12
13     alter = 25; // späte Initialisierung
14     IO.println(name + "(" + geschlecht + "), " + alter + " Jahre, aus " + wohnort);
15 }
```

Konstanten - Übersicht

- Konstanten sind Variablen, die nach der Initialisierung nicht mehr verändert werden können
- Konstanten werden in Java mit dem Schlüsselwort `final` deklariert
- Es wird überprüft, dass keine weitere Zuweisung erfolgt
- Konvention: Konstanten werden in Großbuchstaben geschrieben

Beispieldeklaration und -initialisierung

```
void main() {
    // Deklaration und Initialisierung inkl. Datentyp (Standarfall)
    final String NAME = "Asta Mueller";

    // vereinfachte Deklaration mittels var und Initialisierung (seit Java 10)
    // (Datentyp wird automatisch erkannt)
    final var WOHNORT = "Schönau";
    final var GESCHLECHT = 'd';

    IO.println(NAME + "(" + GESCHLECHT + "), " + " Jahre, aus " + WOHNORT);

    // name = "Berta"; // error: cannot assign a value to final variable name
}
```

Bezeichner (🇺🇸 *Identifier*) - Übersicht

- Bezeichner sind Namen für Variablen, Konstanten, Methoden, Klassen, Interfaces, Enums, etc.
- Erstes Zeichen: Buchstabe, Unterstrich (_) oder Dollarzeichen ($);
- Folgende Zeichen: Buchstaben, Ziffern, Unterstrich oder Dollarzeichen
- Groß- und Kleinschreibung wird unterschieden
- Schlüsselworte (z. B. `var`, `int`, etc.) dürfen nicht als Bezeichner verwendet werden
- Konvention:
  - Variablen (z. B. `aktuellerHerzschlag`) und Methoden (z. B. `println`) verwenden *lowerCamelCase*
  - Konstanten verwenden *UPPER_CASE* und Unterstriche (z. B. `GEWICHT_BEI_GEBURT`)
  - Klassen, Interfaces und Enums verwenden *UpperCamelCase* (z. B. `BigDecimal`)

In Java ist es unüblich, das Dollarzeichen ($) in eigenem Code zu verwenden und es wird in der Regel nur von der JVM (der Java Virtual Machine; d. h. der Ausführungsumgebung) verwendet.

Ein Unterstrich am Anfang des Bezeichners sollte ebenfalls vermieden werden. Ganz insbesondere ist darauf zu verzichten den Unterstrich als alleinigen Variablennamen zu verwenden, da der *reine* Unterstrich seit Java 22 für unbenannte Variablen verwendet wird und dies die Migration von altem Code erschwert.

Übung - Bezeichner

Welche der folgenden Bezeichner sind (a) ungültig, (b) gültig aber sollten dennoch nicht verwendet werden oder (c) gültig und entsprechen den Konventionen?

5.1. Bezeichner

```
var 1a = ...
var 1_a = ...
var _1a = ...
var a1 = ...
```

```
int i;
int _i;
float $$f;
final float E = ...;
```

```
String Wohnort;
String ortDerGeburt;
void BucheFlug(){...}
class FlugBuchungen{...}
```

Übung - Variablen und Konstanten

5.2. Grundlegende Datentypen

- Deklarieren und initialisieren Sie eine Variable x mit dem Ganzzahlwert `42`.
  - Welche Datentypen können Sie verwenden, wenn eine präzise Darstellung des Wertes notwendig ist?
  - Welcher Datentyp wird verwendet, wenn Sie keinen Typ angeben?

    (D. h. wenn Sie `var` schreiben bzw. anders gefragt welchen Typ hat das Literal `42`?)
- Weisen Sie den Wert der Variable `x` einer Variable `f` vom Typ `float` zu.
- Ändern Sie den Wert der Variablen `x`. Welche Auswirkungen hat das auf die Variable `f` vom Typ `float`?
- Deklarieren und initialisieren Sie die Konstante π (Wert `3.14159265359`).
- Deklarieren Sie eine Variable `poem`. Die folgendes - von GitHub Copilot erzeugtes - Gedicht enthält:

```
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.
```

  Achten Sie auf eine konsistente Einrückung/Formatierung des Gedichts.

※ Hinweis

Für diese Aufgabe können Sie sowohl die Java Shell verwenden als auch Ihren Code in eine Datei schreiben. Denken Sie in diesem Fall daran, dass der Code in einer Methode `main` stehen muss (`void main(){ <IHR CODE> }`).

6. Ausdrücke und Operatoren

Ausdrücke und Operatoren - Übersicht

- Berechnungen erfolgen über Ausdrücke, die sich aus Variablen, Konstanten, Literalen, Methodenaufrufen und Operatoren zusammensetzen.
- Jeder Ausdruck hat ein Ergebnis (d. h. Rückgabewert).

  Beispiel: (`age + 1`) addiert zwei Werte und liefert das Ergebnis der Addition zurück.
- Einfache Ausdrücke sind Variablen, Konstanten, Literale und Methodenaufrufe.
- Komplexe Ausdrücke werden aus einfachen Ausdrücken und Operatoren (z. B. +, -, *, /, %, >, <, >=, <=) zusammengesetzt
- Ergebnisse von Ausdrücken können insbesondere Variablen zugewiesen werden (z.B. `int newAge = age + 1` oder `var isAdult = age >= 18`)
- Ausdrücke, die einen Wahrheitswerte ergeben, können zusätzlich in Bedingungen (z. B. `if(age + 5 >= 18) ...`) verwendet werden.

Ausdrücke und Operatoren - Beispiele

```
void main() {
    String s = IO.readln("Enter your age: ");
    int age = Integer.parseInt(s);

    if (age >= 18) {
        IO.println("You are an adult.");
    } else {
        IO.println("You are a minor.");
    }

    var yearsUntil100 = 100-age;
    IO.println("You will be 100 in " + yearsUntil100 + " years.");
}
```

Operatoren und Operanden in der Mathematik

Binäre/Zweistellige Operatoren (🇺🇸 *Binary Operators*)

Addition

$$\begin{array}{ccc} \text{1. Operand} & \text{Operator} & \text{2. Operand} \\ 1 & + & 2 \end{array}$$

Unäre/Einstellige Operatoren (🇺🇸 *Unary Operators*)

Negation

$$\begin{array}{cc} \text{Operator} & \text{Operand} \\ - & (2) \end{array}$$

Fakultät

$$\begin{array}{cc} \text{Operator} & \text{Operand} \\ 2 & ! \end{array}$$

Operatoren

- Operatoren sind spezielle Zeichen, die auf Variablen, Konstanten und Literale angewendet werden, um Ausdrücke zu bilden.
- Die Auswertungsreihenfolge wird durch die Priorität der Operatoren bestimmt.
  (Wie aus der Schulmathematik bekannt gilt auch in Java: `*` oder `/` vor `+` und `-`.)
- Runde Klammern können verwendet werden, um eine bestimmt Auswertungsreihenfolge zu erzwingen bzw. dienen zur Strukturierung
- Es gibt Operatoren, die auf eine, zwei oder drei Operanden angewendet werden: diese nennt man dann ein-, zwei- oder dreistellige Operatoren.
- Für einstellige Operatoren wird die Präfix- oder Postfix-Notation (z.B. `++a` oder `a++`) verwendet,
- Für mehrstellige Operatoren wird die Infix-Notation (z.B. `a + b`) verwendet

Klassifikation der Operatoren

- Arithmetische Operatoren (auf numerische Datentypen)
- Vergleichsoperatoren
- Logische Operatoren (auf boolean Datentypen)
- Bedingungsoperatoren
- Bitoperatoren (auf ganzzahligen Datentypen)
- Zuweisungs- und Verbundoperatoren (auf alle Datentypen)
- Konkatenationsoperator (String)
- Explizite Typkonvertierung

Einige Operatoren sind nur auf bestimmten Datentypen anwendbar. So sind Vergleichsoperatoren wie `<=` oder `>=` nur auf numerische Datentypen anwendbar, aber `==` und `!=` auf allen Typen. Es gilt immer, dass die linke und die rechte Seite Typkompatibel sein müssen; mit anderen Worten wir können nur Dinge vergleichen, die den gleichen Typ haben oder für die eine automatische Typumwandlung möglich ist. Ein Vergleich von einem String und einer Zahl ist z. B. nicht möglich.

Beispiel für unzulässigen Vergleich:

```
jshell> "s" == 1
|  Error:
|  bad operand types for binary operator '=='
|    first type:  java.lang.String
|    second type: int
|  "s" == 1
```

Zweistellige Arithmetische Operatoren

Operator	Anwendung	Bedeutung
`+`	x + y	Summe von x und y (Additions-Operator)
`-`	x - y	Differenz von x und y (Subtraktions-Operator)
`*`	x * y	Produkt von x und y (Multiplikations-Operator)
`/`	x / y	Quotient von x und y (Divisions-Operator)
`%`	x % y	Rest der ganzzahligen Division von x und y (Modulo-Operator)

JShell-Beispiel: ArithmetischeOperatoren.jshell.java

```
1 int x = 3;
2 int y = 5;
3 IO.println( x + y );
4 IO.println( x * y );
5 int z = y / x;
6 IO.println( z );
7 int result = z + z;
8 IO.println( result );
```

6.1. Zweistellige Operatoren - welche Werte werden ausgegeben?

Andere Sprachen (z. B. JavaScript oder Python) haben häufig noch `**` für die Potenzierung. Dies ist in Java über `Math.pow` möglich.

Einstellige Arithmetische Operatoren

Operator	Anwendung	Bedeutung
+	+x	Positiver Wert von x
–	-x	Negativer Wert von x
(Präfix) ++	++x	Prä-inkrement: Gleichbedeutend mit $\{x_{neu} = x_{alt} + 1; x_{neu}\}$
++ (Postfix)	x++	Post-inkrement: Gleichbedeutend mit $\{x_{neu} = x_{alt} + 1; x_{alt}\}$
(Präfix) --	--x	Prä-dekrement: Gleichbedeutend mit $\{x_{neu} = x_{alt} - 1; x_{neu}\}$
-- (Postfix)	x--	Post-dekrement: Gleichbedeutend mit $\{x_{neu} = x_{alt} - 1; x_{alt}\}$

JShell-Beispiel: ArithmetischeOperatoren.jshell.java

```
1 int a = 5;
2 IO.println( ++a );
3 IO.println( a++ );
4 IO.println( -a );
```

6.2. Einstellige Operatoren - welche Werte werden ausgegeben?

Zweistellige Vergleichsoperatoren

Operator	Anwendung	Bedeutung
==	x == y	Überprüft, ob die Werte von x und y **gleich** sind
!=	x != y	Überprüft, ob der Werte von x und y **ungleich** sind
<	x < y	Überprüft, ob der Wert von x **kleiner** dem Wert von y ist
<=	x <= y	Überprüft, ob der Wert von x **kleiner oder gleich** dem Wert von y ist
>	x > y	Überprüft, ob der Wert von x **größer** dem Wert von y ist
>=	x >= y	Überprüft, ob der Wert von x **größer oder gleich** dem Wert von y ist

JShell-Beispiel: Vergleichsoperatoren.jshell.java

```
1  // String-Vergleiche
2  IO.println("Michael" == "Michael");
3  IO.println("Michael" == "michael");
4  IO.println("Michael" != "michael");
5
6  // Vergleiche von numerischen Werten
7  IO.println(1 >= 1);
8  IO.println(2 >= 1d);
9  IO.println(2d >= 3l);
10
11 // UNGÜLTIG: "Michael" == 1
```

6.3. Zweistellige Operatoren - welche Werte werden ausgegeben?

Ein- und zweistellige logische Operatoren

Operator	Anwendung	Bedeutung
!	!x	Negation (Aus true wird false und umgekehrt)
&	x & y	Logisches UND (AND)
&&	x && y	Bedingtes logisches UND (AND Short-circuit Evaluation)
|	x | y	Logisches ODER (OR)
||	x || y	Bedingtes logisches ODER (OR Short-circuit Evaluation)
^	x ^ y	Logisches ENTWEDER-ODER (XOR *exclusive OR*)

Wahrheitstabelle

x	y	!x	x & y oder x && y	x | y oder x || y	x ^ y
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

JShell-Beispiel: LogischeOperatoren.jshell.java

```
1  int x = 5;
2  int y = 7;
3  int n = 0;
4
5  IO.println(x == 5 && y == 7);
6  IO.println(n != 0 && y/n == 1);
7  IO.println(n != 0 & y/n == 1); // ?!
8  IO.println(x == 5 && y/x >= 1);
9
10 IO.println(x == 5 || y/x >= 0);
11 IO.println(x == 5 || y/n >= 0);
12 IO.println(y/n >= 0 || x == 5); // ?!
```

6.4. Logische Operatoren - welche Werte werden ausgegeben?

- Der Unterschied zwischen `&` und `&&` ist, dass `&&` nur den rechten Operanden auswertet, wenn der linke Operand `true` ist.
- Der Unterschied zwischen `|` und `||` ist, dass `||` nur den rechten Operanden auswertet, wenn der linke Operand `false` ist.

Mit anderen Worten bei `&&` und `||` wird der Ausdruck nur so weit ausgewertet, wie nötig ist, um das Ergebnis des Ausdrucks als Ganzes zu bestimmen.

Übung

6.5. Vergleichsoperatoren

Lesen Sie zwei Zahlen von der Console ein (siehe Von der Konsole lesen) und vergleichen Sie diese auf Gleichheit. Speichern Sie das Ergebnis in einer Variable und geben Sie das Ergebnis danach auf der Konsole aus.

Zum Konvertieren der eingelesenen Zeichenketten in Zahlen verwenden Sie die Methode `Integer.parseInt(<EINGABE>)`. Sie können hier den eingelesen String direkt an die Methode übergeben oder vorher in einer Variable speichern.

Denken Sie daran, dass Ihr Code in die `main` Methode gehört:

```
1 void main() {
2     // Ihr Code
3 }
```

Bedingungsoperator

Der Bedingungsoperator:

<Bedingungsausdruck c> "?"
<auszuwertender Ausdruck $a_{(c\ wahr)}$ falls c wahr>
":"
<auszuwertender Ausdruck $a_{(c\ falsch)}$ falls c falsch/unwahr>

liefert in Abhängigkeit eines Ausdrucks c (der einen Wahrheitswert liefert) das Ergebnis des ersten Ausdrucks oder des zweiten Ausdrucks zurück.

$c\ ?\ a_{(c\ wahr)} : a_{(c\ falsch)}$

Beide Ausdrücke $a_{(c\ wahr)}$ und $a_{(c\ falsch)}$ müssen entweder numerische Werte oder boolean Werte oder Instanzen einer Klasse zurück liefern (d. h. Werte die implizit ineinander konvertiert werden dürfen)

Von den beiden Ausdrücken wird *nur ein Ausdruck ausgewertet*.

Beispiele

```
int n = 0;
n == 0 ? 1 : 2
```

Verschachtelung ist möglich aber *nicht* empfehlenswert:

```
int alter = Integer.parseInt(IO.readln("Wie alt sind Sie?"));
alter < 18 ?
    "jugendlicher" :
    alter < 65 ?
        "erwachsener" :
        "senior";
```

Bitoperatoren

Bitoperatoren (>>, <<, ...) arbeiten auf der binären Darstellung der numerischen, primitiven Datentypen für Ganzzahlen.

Bitoperationen werden häufig für spezielle Algorithmen verwendet, um die gleiche Operation auf mehreren Daten (den Bits) gleichzeitig anzuwenden (1 CPU Zyklus).
Ein Beispiel ist das Ver-/Entschlüsseln von Daten (insbesondere mit *XOR*).

Bestimmte mathematische Operationen (z. B. Division durch 2^x) können durch Bitoperationen ersetzt werden, die effizienter sind (z. B. `16 / 4 == 16 >> 2`).

Operator	Anwendung	Bedeutung
~	~x	Bitweise-Negation
&	x & y	Bitweise UND
|	x | y	Bitweise ODER
^	x ^ y	Bitweise ENTWEDER-ODER
<<	x << y	Bits von x werden um y Positionen nach links verschoben und von rechts mit 0 aufgefüllt
>>	x >> y	Bits von x werden um y Positionen nach rechts verschoben und von links mit dem höchsten Bit aufgefüllt
>>>	x >>> y	Bits von x werden um y Positionen nach rechts verschoben und von links mit 0 aufgefüllt

Bits verschieben (*shiften*) um eine bestimmte Anzahl von Positionen:

```
jshell> Integer.toBinaryString(Integer.MIN_VALUE)
$1 ==> "10000000000000000000000000000000"

jshell> Integer.toBinaryString(Integer.MIN_VALUE >> 31)
$2 ==> "11111111111111111111111111111111"

jshell> Integer.toBinaryString(Integer.MIN_VALUE >>> 31)
$3 ==>                                 "1"
```

Verschlüsselung mit XOR (EncryptionWithXOR.jshell.java):

```
final var key = new java.util.Random().nextInt();
Integer.toBinaryString(key);             // ==> "10011010110000111001101010011110"

final var income = 13423;
Integer.toBinaryString(income);          // ==>                  "11010001101111"

// Verschlüsselung von "income" mit "key" mit Hilfe von XOR:
final var encryptedIncome = income ^ key;
Integer.toBinaryString(encryptedIncome); // ==> "10011010110000111111001001000001"
```

Warnung

Die dargestellte Verschlüsselung mit XOR ist die Grundlage aller modernen

Verschlüsselungsalgorithmen, aber es gibt sehr viel zu beachten, um eine sichere Verschlüsselung zu gewährleisten.

Zuweisungs- und Verbundoperatoren

Zuweisungs- und Verbundoperatoren weisen einer Variablen einen neuen Wert zu (z. B. `int newAge = age + 1;`).

Die Variable steht auf der linken Seite des Operators.

Der Ausdruck zur Berechnung des neuen Wertes ist durch den Operator selbst und den Ausdruck auf der rechten Seite festgelegt.

Das Ergebnis des kompletten Ausdruckes ist der zugewiesene Wert mit dem entsprechenden Datentyp.

Standardbeispiele:

```
jshell> int age = 1;
age ==> 1

jshell> age = age + 1;
age ==> 2

jshell> age += 1;
age ==> 3
```

Folgendes wäre auch erlaubt, aber *nicht* empfehlenswert, da schwer(er) zu lesen:

```
jshell> var newAge = age = age + 1;
newAge ==> 4

jshell> var newAge = age += 1;
newAge ==> 5
```

Operator	Bedeutung
x = y	**Zuweisung des Wertes** von y an x
x <Operator>= y	**Zuweisung des Wertes** von x <Operator> y an x

Operatoren: `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `>>>`

Zum Beispiel: `x <<= y` ist gleichbedeutend mit `x = x << y`.

String Konkatenation (Verbinden von Zeichenketten)

Literale, Variablen, Konstanten vom Datentyp String werden durch den Konkatenationsoperator + zu einem neuen String-Wert verkettet.

```
1 jshell> final String name = "Max";
2 name ==> "Max"
3
4 jshell> String greeting = "Hallo " + name + "!";
5 greeting ==> "Hallo Max!"
```

Implizite Typkonvertierung

Bei Zuweisungen und arithmetischen Operationen werden die Datentypen von Operanden unter bestimmten Umständen implizit konvertiert.

- Bei arithmetischen Operationen erfolgt eine Konvertierung in den nächst größeren Datentyp der beteiligten Operanden bzgl. `int`, `long`, `float`, `double`.
- Bei Operationen auf primitiven, ganzzahligen Datentypen wandelt der Compiler die beteiligten Operanden mindestens in `int` um.
- Bei Zuweisungen wird das Ergebnis des Ausdruckes auf der rechten Seite in den Datentyp der Variablen auf der linken Seite konvertiert gemäß der Regeln (Konvertierung von Datentypen).

  ⚠ Die Typkonvertierung erfolgt unabhängig von den konkreten Werten der Operanden.

```
jshell> byte b = 13;
        short s = Short.MAX_VALUE;
        float f = b + s;

b ==> 13
s ==> 32767
f ==> 32780.0
```

```
jshell> int r = Integer.MAX_VALUE + Integer.MAX_VALUE;
r ==> -2
```

⚠ Warnung

Hier erfolgt keine Überlaufprüfung und demzufolge auch keine (implizite) Konvertierung (z. B. in Long).

※ Hinweis

Bei der Addition von `Integer.MAX_VALUE` und `Integer.MAX_VALUE` wird der Wert `-2` zurückgegeben, da der Wert `Integer.MAX_VALUE + 1` den Wert `Integer.MIN_VALUE` ergibt (wir haben einen Überlauf (*Overflow*)).

`Integer.MAX_VALUE + Integer.MAX_VALUE` entspricht also `Integer.MIN_VALUE + (Integer.MAX_VALUE - 1)`.

```
jshell> short s = Short.MAX_VALUE + Short.MAX_VALUE;
|  Error:
|  incompatible types: possible lossy conversion from int to short
|  short s = Short.MAX_VALUE + Short.MAX_VALUE;
```

Explizite Typkonvertierung

Das Ergebnis eines Ausdruckes kann durch explizite Typkonvertierung in einen anderen primitiven Datentyp umgewandelt werden.

- Bei primitiven Datentypen erlaubt für numerische Datentypen.
- Wird ein ganzzahliges Ergebnis in einen kleineren ganzzahligen Datentyp konvertiert, dann werden die führenden Bits abgeschnitten.
- Nachkommastellen gehen bei der Konvertierung von Gleitkommazahlen in Ganzzahlen verloren
- Bei Konvertierung von `double` in `float` kommt es ebenfalls zu einem Genauigkeitsverlust in der Darstellung (durch Abschneiden der Bits in Mantisse und Exponent)

Standardfälle

```
jshell> int i = 42;
i ==> 42

jshell> byte b = (byte) i;
b ==> 42
```

Sonderfälle

```
jshell> (byte) 128 ;
$1 ==> -128

jshell> (byte) 256 ; // Integer.numberOfTrailingZeros(256) == 8
$2 ==> 0
```

Überlauf und Unterlauf

Unter-/Überschreitet das Ergebnis eines Ausdruckes den minimalen/maximalen Wert des resultierenden Datentyps, erfolgt ein Unter-/Überlauf. (🏴 *Overflow*/🏴 *Underflow*)

- Bei einem Unterlauf bzw. Überlauf werden bei Ganzzahlen die nicht mehr darstellbaren höheren Bits abgeschnitten.
- Bei Fließkommazahlen werden die Konstanten: `Float.NEGATIVE_INFINITY` und `Float.POSITIVE_INFINITY` bzw. `Double.NEGATIVE_INFINITY` und `Double.POSITIVE_INFINITY` verwendet.

```
Integer.toBinaryString(Integer.MIN_VALUE) //    "10000000000000000000000000000000"
Integer.toBinaryString(Integer.MIN_VALUE -1) // "01111111111111111111111111111111"
Long.toBinaryString(Integer.MIN_VALUE -1l)
             // "11111111111111111111111111111101111111111111111111111111111111"
```

In der Praxis wird häufig der Begriff Overflow verwendet, wenn bei einer Berechnung der Wertebereich eines Datentyps nicht ausreicht, um das Ergebnis zu speichern.
D. h. die Unterscheidung zwischen Über- und Unterlauf ist nicht immer eindeutig.

Bei Double erfolgt der Überlauf erst, wenn man eine Zahl auf `Double.MAX_VALUE` addiert, die mind. 292 Stellen vor dem Komma hat.

```
jshell> Double.MAX_VALUE + 1
$0 ==> 1.7976931348623157E308

jshell> Double.MAX_VALUE + 899999999999999999999999
99999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999
9999999d // Acht gefolgt von 291 Neunen(!)
$1 ==> 1.7976931348623157E308 // 16 Nachkommastellen

jshell> Double.MAX_VALUE + 999999999999999999999999
99999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999
9999999d // 292 Neunen(!)
$2 ==> Infinity
```

Auswertungsreihenfolge

Die Auswertungsreihenfolge von komplexen Ausdrücken mit mehreren Operatoren wird durch die Priorität der Operatoren bestimmt.[4]

- Kommen in einem Ausdruck mehrere Operatoren mit gleicher Priorität vor, dann wird der Ausdruck von links nach rechts ausgewertet.
- Ausnahmen sind die Verbund- und Zuweisungsoperatoren die von rechts nach links bewertet werden.
- Klammern haben die höchste Priorität und erzwingen die Auswertung des Ausdrucks in den Klammern zuerst. Klammern dienen aber (insbesondere) auch der Strukturierung von Ausdrücken.

---

[4] Die Regeln sind vergleichbar mit der Schulmathematik: Punkt-vor-Strich-Rechnung.

Priorität der Operatoren

Operatoren	Beschreibung	Priorität
=, +=, -=, ...	Zuweisungs- und Verbundoperatoren	1 (niedrigste)
?:	Bedingungsoperator	2
||	Bedingt logisches ODER	3
&&	Bedingt logisches UND	4
|	Logisches/Bitweises ODER	5
^	Logisches/Bitweises ENTWEDER-ODER	6
&	Logisches/Bitweises UND	7
==, !=	Vergleich: Gleich, Ungleich	8
<, <=, >, >=	Vergleich: Kleiner (oder Gleich), Größer (oder Gleich)	9
<<, >>, >>>	Bitweise Schiebeoperatoren	10
+, -	Addition, Subtraktion, String-Konkatentation	11
*, /, %	Multiplikation, Division, Rest	12
++, --, +, - (Vorzeichen), ~, !, (cast)	Einstellige Operatoren	13 (höchste)

Übung zur Auswertungsreihenfolge

6.6. Auswertung von Ausdrücken

Sind die folgenden Ausdrücke (a) gültig und wie ist (b) ggf. das Ergebnis der folgenden Ausdrücke und (c) welchen Wert haben die Variablen nach der Auswertung?

(*Achtung: der neue Wert wird dann für den nachfolgenden Ausdruck verwendet.*)

Initiale Belegung der Variablen: `int x = 4, y = 2, z = 3;`.

```
1  x + y * z / x
2  ( x + - (float) y * 2 ) / x  ==  ( x + ( ( (float) -y ) * 2 ) )/ x
3  x + ++y * z++ % x
4  x < 5 && --y <= 1 || z == 3
5  x << 2 * y >> 1
6  z & 1 % 2 == 0
7  (z & 1) % 2 == 0
```

Übung

6.7. Umrechnung von Sekunden

Schreiben Sie ein Java-Skript, dass die Anzahl von Sekunden in Stunden, Minuten und Sekunden umrechnet. Lesen Sie die Anzahl von Sekunden von der Konsole ein und geben Sie die Umrechnung auf der Konsole aus.

Beispiel

```
Bitte geben Sie die Sekunden ein: 3455
0 Stunde(n), 57 Minute(n) und 35 Sekunde(n)
```

Denken Sie daran, dass Ihr Code in die `main` Methode gehört:

```
1 void main() {
2     // Ihr Code
3 }
```

Denken Sie daran, dass Sie einen Zeichenkette (`String`) in eine Zahl umwandeln können, indem Sie die Methode `Double.parseDouble(<String>)` für Fließkommazahlen verwenden oder `Integer.parseInt(<String>)` für Ganzzahlen.

Schreiben Sie ein vollständiges Java-Skript, dass Sie mit dem Java Interpreter (`java --enable-preview <JAVA-DATEI>`) ausführen können.

Übung

6.8. Body-Mass-Index (BMI) berechnen mit Java (Skript)

Lesen Sie das Gewicht in Kilogramm und die Größe in Metern von der Konsole ein und geben Sie den BMI auf der Konsole aus. Geben Sie auch aus, ob die Person Untergewicht (`BMI < 18,5`), Normalgewicht oder Übergewicht (`BMI >= 25`) hat. Falls die Person nicht das Normalgewicht hat, geben Sie auch an, wie viel Gewicht sie bis zum Normalgewicht zunehmen oder abnehmen muss. Berechnungsvorschrift: $BMI = \frac{Gewicht}{Größe^2}$.

Benutzen Sie nur die Konstrukte, die Sie bisher gelernt haben!

Beispiel

```
Bitte geben Sie Ihr Gewicht in kg ein: 85
Bitte geben Sie Ihre Größe in m ein: 1.80
Ihr BMI beträgt: 26.234567901234566
Untergewicht: nein
Normalgewicht: nein
Übergewicht: -4.0 kg bis Normalgewicht
```

Von Variablen, Konstanten, Literalen und Ausdrücken

- Variablen sind Speicherorte, die einen Wert enthalten.
- Konstanten sind unveränderliche Werte, die an einem Speicherort gespeichert sind.
- Literale sind konstante Werte, die direkt im Code stehen.
- Operatoren haben eine Priorität und bestimmen die Auswertungsreihenfolge von Ausdrücken.
- Ausdrücke sind Kombinationen von Variablen, Konstanten und Operatoren, die einen Wert ergeben.
- Implizite Typkonvertierung erfolgen automatisch und führen meist zu keinem Verlust von Genauigkeit.

7. (Bedingte) Anweisungen, Schleifen und Blöcke

Anweisungen

Eine Anweisung in einem Java-Programm stellt eine einzelne Vorschrift dar, die während der Abarbeitung des Programms auszuführen ist.

- In Java-Programmen werden einzelne Anweisungen durch einen Semikolon `;` voneinander getrennt.

```
1 void main() {
2     int a = 1; // Variablendeklaration und Initialisierung
3     IO.println("a = " + a); // Methodenaufruf (hier: println)
4 }
```

- Programme setzen sich aus einer Abfolge von Anweisungen zusammen.
- Die einfachste Anweisung ist die leere Anweisung: `;`.
- Weitere Beispiele für Anweisungen sind Variablendeklarationen und Initialisierungen, Zuweisungsausdrücke, Schleifen, Methoden-Aufrufe.

Blöcke

Ein Block in einem Java-Programm ist eine Folge von Anweisungen, die durch geschweifte Klammern { ... } zusammengefasst werden.

- Blöcke werden *nicht* durch einen Semikolon beendet.

```
1 void main() {
2     {   // Block von Anweisungen
3         int a = 1;
4         IO.println("a = "+a);
5     }
6 }
```

- Ein Block kann dort verwendet werden, wo auch eine Anweisung erlaubt ist.
- Ein Block stellt ein Gültigkeitsbereich ( *scope*) für Variablendeklarationen dar. Auf die entsprechenden Variablen kann nur von innerhalb des Blocks zugegriffen werden.
- Leere Blöcke {} sind erlaubt und Blöcke können verschachtelt werden.

Anweisungen und Blöcke - Beispiele

```java
// Deklaration und Initialisierung von Variablen
int age = 18 + 1;
char gender = 'm';

; // Leere Anweisung

// Block
{
    boolean vegi = true;
    gender = 'f';
    IO.println("vegi=" + vegi);
    {} // leerer Block
}

// Methodenaufruf
IO.println("age=" + age);
IO.println("gender=" + gender);
/* IO.println("vegi=" + vegi); ⇒  Error: cannot find symbol: variable vegi */
```

Bedingte Anweisungen und Ausdrücke

Bedingte Anweisungen und Ausdrücke in einem Java-Programm dienen dazu Anweisungen bzw. Blöcke nur dann auszuführen wenn eine logische Bedingung eintrifft.

- Bedingte Anweisungen und Ausdrücke zählen zu den Befehlen zur Ablaufsteuerung.
- Bedingte Anweisungen und Ausdrücke können in Java-Programmen mittels `if`-Anweisungen, `if`-/`else`-Anweisung und `switch`-Anweisungen/-Ausdrücken umgesetzt werden.
- Der Bedingungs-Operator (`<Ausdruck> ? <Ausdruck> : <Ausdruck>`) stellt in bestimmten Fällen eine Alternative zu den bedingten Anweisungen dar.

`if`-Anweisung

Die `if`-Anweisung setzt sich zusammen aus dem Schlüsselwort `if`, einem Prüf-Ausdruck in runden Klammern und einer Anweisung bzw. einem Block.

Syntax: `if(<Ausdruck>) <Anweisung> bzw. <Block>`

```
1  void main() {
2      var age = Integer.parseInt(IO.readln("Wie alt sind Sie?"));
3      boolean adult = false;
4
5      if (age >= 18) { // if-Anweisung
6          adult = true;
7      }
8
9      IO.println("adult=" + adult);
10 }
```

```
1  void main() {
2      var age = Integer.parseInt(IO.readln("Wie alt sind Sie?"));
3      var adult = false;
4      char status = 'c';
5
6      if (age >= 18) {
7          adult = true;
8          status = 'b';
9          if (age >= 30 && IO.readln("Geschlecht (m/w/d)?").charAt(0) == 'm')
10             status = 'a';
11     }
12     IO.println("adult=" + adult+ ", status=" + status);
13 }
```

Der `<Ausdruck>` muss einen Wert vom Datentyp `boolean` zurückliefern

Die `<Anweisung>` bzw. der `<Block>` wird ausgeführt, wenn der Ausdruck `true` zurück liefert

Ansonsten wird die nächste Anweisung nach der `if`-Anweisung ausgeführt

`if`-Anweisungen können verschachtelt werden (in der Anweisung bzw. im Block).

if-else-Anweisung

```
1  void main() {
2      IO.println("Anzahl der Tage in einem Monat");
3
4      var month = Integer.parseInt(IO.readln("Welchen Monat haben wir(1-12)? "));
5
6      int days = 31;
7      if (month == 2 && IO.readln("Schaltjahr (j/n)? ").charAt(0) == 'j') days = 29;
8      else if (month == 2)
9          days = 28;
10     else if (month == 4)
11         days = 30;
12     // ...
13     IO.println("days=" + days);
14 }
```

```
1  void main() {
2      IO.println("Anzahl der Tage in einem Monat");
3
4      var month = Integer.parseInt(IO.readln("Welchen Monat haben wir(1-12)? "));
5
6      int days = 31;
7      if (month == 2 && IO.readln("Schaltjahr (j/n)? ").charAt(0) == 'j') days = 29;
8      else if (month == 2) days = 28; // "nur" umformatiert
9      else if (month == 4) days = 30;
10     // else ...
11     IO.println("days=" + days);
12 }
```

Die `if`-Anweisung kann um einen `else`-Zweig erweitert werden, der aus dem Schlüsselwort `else` und einer Anweisung bzw. einem Block besteht.

Syntax: `if(<Ausdruck>) <Anweisung bzw. Block> else <Anweisung bzw. Block>`

Die `<Anweisung>` bzw. der `<Block>` im else-Zweig wird ausgeführt, wenn der Ausdruck in der `if`-Anweisung `false` zurück liefert.

Im `else`-Zweig kann wieder eine weitere `if`-Anweisung verwendet werden (`if` / `else-if` Kaskade).

Bei verschachtelten `if`-Anweisungen gehört der `else`-Zweig zur direkt vorhergehenden `if`-Anweisung ohne `else`-Zweig.

Übung

7.1. Berechnung des BMI mit if-else Anweisung

Stellen Sie die vorherige Lösung zum Berechnen des BMI so um, dass Sie nur `if`-`else`-Anweisungen verwenden und keinen Bedingungsoperator (`? :`).

`switch`-Anweisung/-Ausdruck (Grundlagen)

Die `switch`-Anweisung bzw. der `switch`-Ausdruck setzt sich aus dem Schlüsselwort `switch`, einem Prüf-Ausdruck in runden Klammern und einem oder mehreren `case`-Blöcken zusammen.

Syntax: `switch(<Ausdruck>) <case-Block>* [<default-Block>]`

Im Gegensatz zur `if-else` Anweisung wird hier nur ein <Ausdruck> ausgewertet für den mehrere Alternativen (`case`-Blöcke) angegeben werden können.

Der `default`-Zweig stellt eine Möglichkeit dar, die immer dann ausgeführt wird, wenn kein anderer `case`-Block zutrifft

Syntax: `default: <Anweisungen>`

`case L :`

Syntax: `case <Literal>: <Anweisungen>`.

- Ein `case`-Block setzt sich zusammen aus dem Schlüsselwort `case`, einem oder mehreren `Literal`en (konstanter Ergebniswert) und einer Abfolge von Anweisungen.
- Die Anweisung in einem `case :`-Block werden bis zur folgenden `break`-Anweisung ausgeführt (*fall-through*).
- Gibt es keine `break`-Anweisung in einem `case`-Block werden alle Anweisungen bis zum Ende der `switch`-Anweisung ausgeführt.
- Für `switch`-Ausdrücke ist am Ende eines `case`-Blocks eine `yield`-Anweisung erforderlich, die den Wert des `case`-Blocks zurückliefert.

```java
void main() {
    var month = Integer.parseInt(IO.readln("Welchen Monat haben wir(1-12)? "));
    int days = 31;
    switch (month) { // seit Java 1.0
        case 2:
            if (IO.readln("Schaltjahr (j/n)? ").charAt(0) == 'j') days = 29;
            else days = 28;
            break;
        case 4:
        case 6:
        case 9:
        case 11: // ⩽ possible, but unusual formatting
            days = 30;
            break;
    }
    IO.println("Anzahl der Tage im Monat " + days);
}
```

```java
void main() {
    var month = Integer.parseInt(IO.readln("Welchen Monat haben wir(1-12)? "));
    int days = 31;
```

```
    switch (month) { // seit Java 14
        case 2:
            if (IO.readln("Schaltjahr (j/n)? ").charAt(0) == 'j') days = 29;
            else days = 28;
            break;
        case 4, 6, 9, 11:
            days = 30;
            break;
    }
    IO.println("Anzahl der Tage im Monat " + days);
}
```

```
void main() {
    var month = Integer.parseInt(IO.readln("Welchen Monat haben wir(1-12)? "));
    // seit Java 14:
    int days =
        switch (month) { // Switch-Ausdruck
            case 2:
                yield IO.readln("Schaltjahr (j/n)? ").charAt(0) == 'j' ? 29 : 28;
            case 4:
            case 6, 9, 11:
                yield 30;
            default:
                yield 31;
        };
    IO.println("Anzahl der Tage im Monat " + days);
}
```

case L →

Syntax: `case <Literal> → <Ausdruck oder Block>.`

- Auf der rechten Seite ist nur ein Ausdruck oder ein Block erlaubt - keine Anweisung.
- Bei dieser Variante gibt es kein *durchfallen* (*Fall-Through*), d. h. ein `break` ist nicht zur Beendigung eines `case`-Blocks zu verwenden!

```
void main() {
    var month = Integer.parseInt(IO.readln("Welchen Monat haben wir(1-12)? "));
    int days = 31;
    switch (month) { // seit Java 14
        case 2 -> { // Block oder Ausdruck!
            if (IO.readln("Schaltjahr (j/n)? ").charAt(0) == 'j') days = 29;
            else days = 28;
        }
        case 4, 6, 9, 11 -> days = 30;
    }
    IO.println("Anzahl der Tage im Monat " + days);
}
```

```
void main() {
    var month = Integer.parseInt(IO.readln("Welchen Monat haben wir(1-12)? "));
    // seit Java 14
    int days =
        switch (month) { // Switch-Ausdruck
```

```
6               case 2 -> IO.readln("Schaltjahr (j/n)? ").charAt(0) == 'j' ? 29 : 28;
7               case 4, 6, 9, 11 -> 30;
8               default -> 31;
9           };
10      IO.println("Anzahl der Tage im Monat " + days);
11  }
```

Als Wert im `case`-Block können Literale vom Datentyp `int` und ab Java 7 auch `String` und Aufzählungen (`enum` Klassen) verwendet werden; ab Java 21 wird der Musterabgleich (*pattern matching*) unterstützt es können auch beliebige (sogenannte) Referenztypen (nicht nur `String`) verwendet werden. Wir werden dies später bei der Diskussion von Referenztypen detailliert behandeln.

`case L ->` wird erst seit Java 14 unterstützt. Ein Mischen ist nicht möglich.

`switch`-Anweisung ≙ *switch-statement*

`switch`-Ausdruck ≙ *switch-expression*

`switch`-Anweisung/-Ausdruck mit Musterabgleich und `when` Bedingungen (seit Java 21)

Seit Java 21 werden auch `case`-Label unterstützt, die Muster abgleichen ( *match a pattern*), und die mit `when`-Bedingungen kombiniert werden können.

Syntax: `case <Pattern> when <Bedingung> -> <Ausdruck oder Block>;`

```
void main() {
    var name = IO.readln("Wie ist Dein Name? ");
    String nameAnalysis = switch (name) {
        // Der erste Vergleich, der zutrifft, wird ausgeführt.
        case "Michael", "Tom", "Erik"               -> "m";
        case "Alice", "Eva", "Maria", "Eva-Maria"   -> "w";
        case String s when s.length() < 2           -> "kein Name";
        case _ when name.contains("-")              -> "Doppelname";
        default                                     -> "<unbekannt>";
    };
    IO.println("Namensanalyse = " + nameAnalysis);
}
```

7.2. Erfolgreicher Musterabgleich?

Bei welchem Name wäre ein erfolgreicher Musterabgleich in mehreren Fällen möglich?

Wir werden Pattern Matching später detailliert behandeln.

Effizienz von bedingten Anweisungen

- Bei `if`-/`else`-Anweisungen werden die Prüf-Ausdrücke sequentiell (in der angegebenen Reihenfolge) ausgewertet (ein Ausdruck pro Alternative).
- Bei (klassischen) `switch`-Anweisungen/-Ausdrücken wird nur ein einziger Prüf-Ausdruck ausgewertet und die entsprechende(n) Alternative(n) direkt oder zumindest sehr effizient ausgeführt.
- Daher benötigt die Auswertung einer `switch`-Anweisung i. d. R. weniger Rechenschritte als eine äquivalente `if`-/`else`-Anweisung.

Übung

7.3. Wochentag benennen

Lesen Sie (a) den Tag des Monats, (b) den Monat, (c) ob das Jahr ein Schaltjahr ist und (d) den Wochentag des 1. Januars des Jahres ein. Benutzen Sie `switch` und/oder `if`-Anweisungen und geben Sie den Wochentag des gegebenen Datums aus.

Beispiel

```
# java Wochentag.java
Welchen Monat haben wir (1-12)? 12
Welchen Tag des Monats haben wir (1-28/29/30/31)? 24
Welcher Wochentag war der 1. Januar (0=Montag, 1=Dienstag, ..., 6=Sonntag)? 0
Ist das Jahr ein Schaltjahr (j/n)? j
> Tag im Jahr: 359
> Tag in der Woche: 2
> Der 24.12. ist ein Dienstag
```

Optional: Erlauben Sie statt der Eingabe einer Zahl für den Wochentag auch die Eingabe des Wochentages als Text (z. B. „Montag“, „Dienstag“, ...).

8. Schleifen

Schleifen

- Schleifen dienen dazu gleiche Anweisungen bzw. Blöcke mehrfach auszuführen
- Schleifen zählen wie auch bedingte Anweisungen zu den Befehlen der Kontrollflußsteuerung
- Schleifen können in Java-Programmen mittels for-Anweisungen, while-Anweisung und do-while-Anweisungen umgesetzt werden
- Es muss darauf geachtet werden, dass keine Endlosschleifen entstehen

for-Schleife

```
int sum = 0;

// for(<Init>; <Ausdruck>; <Update>) <Anweisung>
for(int i = 0; i < 10 ; ++i){
    sum += i;
    IO.println("sum="+sum);
}
```

Die for-Schleife setzt sich zusammen aus einer Initialisierungsliste (<Init>), einer Abbruchbedingung <Ausdruck>, einer Änderungsliste (<Update>) und einen Schleifenrumpf (<Anweisung> bzw. <Block>). Alle drei Teile sind optional.

Syntax: `for(<Init>;<Ausdruck>;<Update>) <Anweisung> bzw. <Block>`

Initialisierungsliste:
wird vor dem ersten evtl. Schleifendurchlauf ausgeführt

Abbruchbedingung:
wird vor jedem Schleifendurchlauf geprüft

Änderungsliste: wird nach einem Schleifendurchlauf ausgeführt

Sowohl die Initialisierungsliste als auch die Änderungsliste können mehrere Ausdrücke enthalten, die durch Kommas getrennt sind.

Beispiel:

```
int sum=0;


for(int i=0, j=2; i < 10; ++i, j+=2){
    sum +=j;
    IO.println("sum="+sum);
}
```

```
int sum=0;
int i=-1;
int j=3;
for(i++, j--; i < 10 ; ++i, j+=2){
    sum +=j;
    IO.println("sum="+sum);
}
```

Gültiger Code:

```
for(;;) { IO.println("forever"); }
```

while-Schleife

```
1  int sum = 0;
2  int i = 0;
3
4  // while(<Ausdruck>) <Anweisung oder Block>
5  while(i < 10){
6      sum += i;
7      IO.println("sum=" + sum);
8      ++i;
9  }
```

Die while-Anweisung setzt sich zusammen aus einem <Ausdruck> als Abbruchbedingung und einen Schleifenrumpf (<Anweisung> bzw. <Block>).

Syntax: while(<Ausdruck>) <Anweisung> bzw. <Block>

Die Abbruchbedingung wird vor jedem Schleifendurchlauf geprüft.

do-while-Schleife

```
1  int sum=0;
2  int i=0;
3
4  // do <Anweisung> while (<Ausdruck>);
5  do {
6      sum += i++;
7      IO.println("sum="+sum);
8  } while(i < 10);
```

- Die do-while-Schleife setzt sich zusammen aus einem Schleifenrumpf (<Anweisung> bzw. <Block>) und einem <Ausdruck> als Abbruchbedingung.
- Die Abbruchbedingung wird nach jedem Schleifendurchlauf geprüft.

  Im Gegensatz zur while-Schleife wird der Schleifenrumpf mindestens einmal ausgeführt, bevor die Abbruchbedingung geprüft wird.

Kontrolle des Schleifenablaufs

```java
int sum = 0;

for (int i = 0; i < 10; ++i) {
    if ((i + 1) % 5 == 0)
        break;

    sum += i;
    IO.println("i=" + i);
}
IO.println("sum=" + sum);
```

```java
int sum = 0;

for (int i = 0; i < 10; ++i) {
    if ((i + 1) % 5 == 0)
        continue;

    sum += i;
    IO.println("i=" + i);
}
IO.println("sum=" + sum);
```

```java
int sum = 0;

outer: for (int i = 0; i < 10; ++i) {
    IO.println("i=" + i);

    for (int j = 0; j < i; ++j) {
        IO.println("j=" + j);
        if ((j + 1) % 5 == 0)
            break outer;
        sum += j;
    }

    sum += i;
}
IO.println("sum=" + sum);
```

- Mit den Anweisungen `break`, `break <Marke>`, `continue` und `continue <Marke>` kann die Abarbeitung einer Schleife beeinflusst werden.
- Bei `break` wird die Ausführung des aktuellen Schleifendurchlaufs abgebrochen und mit der Anweisung direkt nach dem Schleifenrumpf fortgefahren.
- Bei `continue` wird die Ausführung des aktuellen Schleifendurchlaufs abgebrochen und zum nächsten Schleifendurchlauf gesprungen.
- `break <Marke>` bricht auch die Ausführung des aktuellen Schleifendurchlaufs ab und es wird zur Anweisung nach einem Schleifenrumpf der Schleife mit der gegebenen Marke gesprungen.
- Eine Marke setzt sich zusammen aus einem Java-Bezeichner und einem „:“ und

kann vor einer Schleife bzw. einem Block stehen.

Corner Cases

```
jshell> farOuter: for (int j = 0 ; j < 5 ; j++)
            outer: for(int i = 0; i < 5; i++) {
                IO.println(j+" "+i); break farOuter;
            }
0 0

jshell> farOuter: for (int j = 0 ; j < 5 ; j++)
            outer: for(int i = 0; i < 5 ; i++) {
                IO.println(j+" "+i); continue farOuter;
            }
0 0
1 0
2 0
3 0
4 0
```

Übung

8.1. Einfacher Primzahltest

Verwenden Sie eine Schleife, um festzustellen ob eine Zahl eine Primzahl ist. Lesen Sie die Zahl von der Konsole ein. Geben Sie am Ende aus, ob die Zahl eine Primzahl ist oder nicht; geben Sie ggf. auch den kleinsten Teiler der Zahl aus.

- Schreiben Sie den Code für den Java Interpreter.
- Es ist nicht erforderlich, dass der Algorithmus effizient ist.

Beispiel

```
# java Primzahltest.java
Geben Sie eine ganze positive Zahl ein? 97
97 ist eine Primzahl
# java Primzahltest.java
Geben Sie eine ganze positive Zahl ein? 123
3 ist ein Teiler von 123
```

Übung

8.2. Berechnung der Fibonacci Zahlen

Schreiben Sie ein Programm, das die n-te Fibonacci-Zahl berechnet und auf der Konsole ausgibt. Lesen Sie n von der Konsole ein.

Die Fibonacci-Zahlen sind definiert durch die Rekursionsformel F(n) = F(n-1) + F(n-2) mit den Anfangswerten F(0) = 0 und F(1) = 1.

Die ersten 10 Fibonacci Zahlen:

F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	F(8)	F(9)	F(10)
0	1	1	2	3	5	8	13	21	34	55

Beispiel

```
# java --enable-preview Fibonacci.java
Welche Fibonacci Zahl möchten Sie berechnen? 10
55
```

Übung

8.3. Berechnung der Kubikwurzel mit Newton-Raphson

Berechnen Sie die Kubikwurzel x einer Zahl n mit Hilfe einer Schleife. Nutzen Sie dazu das schnell konvergierende, iterative Verfahren von Newton-Raphson.

$$x_{n+1} = x_n - \frac{x_n^3 - n}{3 \times x_n^2} \qquad \text{ein mgl. Startwert: } x_0 = 1$$

Beispiel

```
# java KubikwurzelMitSchleife.java  10:43:30
Geben Sie eine Zahl n ein deren Kubikwurzel w Sie berechnen wollen
(d.h. n = w*w*w): 1000000
Wie viele Schritte wollen Sie machen? 50
...
Das Ergebnis ist: 100.0
```

Stellen Sie sich die folgenden Fragen:

- Welchen Datentyp sollten Sie für die Kubikwurzeln verwenden?
- Macht es Sinn die Anzahl der Iterationen zu begrenzen?
  (D. h. wie schnell konvergiert das Verfahren?)
- Können Sie die Kubikwurzel von 2.251.748.274.470.911 (2_251_748_274_470_911) berechnen?
- Wie kann man feststellen ob eine gute Näherung an die Kubikwurzel vorliegt?

Um zu verstehen wie schnell der Algorithmus konvergiert können sie sich den aktuellen Wert x_n ausgeben lassen.

Hausübung: Implementieren Sie den Algorithmus auch mit einem anderen Typ von Schleife.

9. Methoden

Methoden (in Java-Skripts)

- Methoden in Java-Programmen dienen dazu die Anwendungslogik zu strukturieren und in wiederverwendbare *Unterprogramme* zu zerlegen.
- Methoden können von einer anderen Methode aufgerufen werden.
- Eine Methode hat einen Namen, eine Parameterliste und einen Rückgabetyp.

  Methoden können bzw. müssen weiterhin deklarieren welche Ausnahmen auftreten können. Dies werden wir aber erst später behandeln.
- Der Methodenrumpf ist eine Abfolge von Anweisungen bzw. Blöcken.

Syntax:

```
<Rückgabetyp> <Methodenname> (<Parameterliste>){
    <Methoden-Rumpf>
}
```

- Wir haben bereits Methoden wie `println(String)` und `double Double.parseDouble(String)` kennengelernt.
- Wenn wir `void main() { ... }` verwenden, dann definieren wir eine Methode, die von der Java-Laufzeitumgebung beim Start aufgerufen wird.

  Per Konvention ist festgelegt, dass diese Methode `main` heisst.

Beispiel

Deklaration einer Methode zum Berechnen des größten gemeinsamen Teilers (`ggt`) zweier Zahlen.

```
1  int ggt(int z1, int z2) { // Algorithmus von Euklid
2      z1 = Math.abs(z1);
3      z2 = Math.abs(z2);
4
5      int rest = 0;
6      while (z2 != 0) {
7          rest = z1 % z2;
8          z1 = z2;
9          z2 = rest;
10     }
11     return z1; // return bestimmt welcher Wert zurückgegeben wird
12 }
```

Im `ggtEuklid`-Beispiel werden die Parameter als normale Variablen behandelt. Dies wurde hier aus Kompaktheitsgründen so gewählt. Im Allgemeinen sollten die Parameter als Konstanten betrachtet werden, d. h. sie sollten nicht verändert werden.

Methodenparameter und Rückgabewerte

- Die Parameterliste definiert über eine komma-separierte Liste die optionalen formalen Parameter der Methode:

  Syntax: `<Typ> <Bezeichner> (, <Typ> <Bezeichner>)*`

- Rückgabewerte werden im Methodenrumpf mit der return Anweisung zurückgegeben:

  Syntax: `return <Ausdruck>`

- Bei *Methoden ohne Rückgabewert* (`void`) dient die leere `return` Anweisung (`return ;`) zum - ggf. vorzeitigem - Beenden der Methode. Am Ende der Methode ist in diesem Fall die `return` Anweisung optional.

Methodenaufruf

- Der Aufruf erfolgt durch die Angabe des Klassennamens, des Methodennamens und der aktuellen Parameterwerte.

  Syntax: <Methode>(<Parameterwerte>)

- Als aktuelle Parameterwerte können Variablen, Ausdrücke oder Literale übergeben werden.
- Der Datentyp des übergebenen Wertes muss in den Datentyp des formalen Parameters implizit konvertierbar sein. Andernfalls muss explizit konvertiert werden.
- Von allen übergebenen Werten wird eine (ggf. flache) Kopie übergeben.

  D. h. Änderungen an den Parametern innerhalb der Methode haben keine Auswirkungen auf die Werte der Argumente (Fachbegriff: *call-by-value*).
- Methoden dürfen sich selber aufrufen (**Rekursion**).

Rekursive Methoden - Beispiel

Schleifen basierte Implementierung des Algorithmus von Euklid:

```
int ggt(int z1, int z2) {
    int rest = 0;
    while (z2 != 0) {
        rest = z1 % z2;
        z1 = z2;
        z2 = rest;
    }
    return z1;
}
```

Elegante rekursive Implementierung des Algorithmus von Euklid:

```
int ggt(int z1, int z2) {
    if (z2 == 0)
        return z1;
    else
        return ggt(z2, z1 % z2);
}
```

Achtung!

In vielen Programmiersprachen (inkl. Java) ist die Rekursion in _bestimmten_ - aber häufigen - Fällen nicht so effizient wie Schleifen.

Überladen von Methoden (🇺🇸 *Overloading*)

- Eine überladene Methode ist eine Methode mit dem gleichen Namen wie eine andere Methode, aber mit einer unterschiedlichen Parameterliste. Folgende Unterschiede sind möglich:
- Eine Methode definiert eine unterschiedliche Anzahl von Parametern
- Eine Methode hat unterschiedliche Datentypen für ihre formalen Parameter
- Unterschiedliche Rückgabetypen sind in Java nicht ausreichend.
- Zum Beispiel gibt es in Java die Methode `int Math.max(int, int)` und `double Math.max(double, double)`.

9.1. `max(long, long)`?

Warum definiert Java auch noch die Methode `long Math.max(long, long)` bzw. warum reicht es nicht aus nur `long Math.max(long, long)` zu definieren und auf `int Math.max(int, int)` zu verzichten?

```
1  void print(int i) {
2      IO.println("int: " + i);
3  }
4
5  void print(double d) {
6      IO.println("double: " + d);
7  }
8
9  void main() {
10     int i = 1;
11     IO.print(i);
12     float f = 1.0f;
13     IO.print(f);
14 }
```

Aufruf von Methoden aus anderen Klassen

- *Für den Moment* ist eine Klasse für uns eine Sammlung von Methoden und Konstanten, die inhaltlich in einem logischen Zusammenhang stehen.
- Der Aufruf einer sogenannten Klassenmethode (solche mit dem Modifizierer `static`) einer Klasse erfolgt durch die Angabe des Klassennamens, des Methodennamens und der aktuellen Parameterwerte.

  Syntax: <Klasse>.<Methode>(<Parameterwerte>)

- Wir haben bereits entsprechende Beispiele gesehen, z. B. `Double.parseDouble(String)` oder `Integer.parseInt(String)`.

※ Hinweis

Auf diese Weise können nur statische Methoden aufgerufen werden. Die Details werden wir später beim Thema Klassen und Objekte behandeln.

Übung

9.2. Methoden definieren

1. Nehmen Sie die Ergebnisse der letzten Übung und definieren Sie jeweils eine Methode für die Berechnung der `Kubikwurzel` und für den Primzahltest. Die Methode `isPrime` soll dabei den Rückgabetyp boolean haben.

   Auf die Ausgabe des kleinsten Teilers beim Primzahltest soll verzichtet werden.

2. Rufen Sie die Methoden aus Ihrer `main`-Methode auf. Die `main`-Methode soll dabei nur die Eingabe und die Ausgabe übernehmen.
3. Wandeln Sie die Methode für die Berechnung der Kubikwurzel in eine rekursive Methode um.

Übung

9.3. Fakultät berechnen

Schreiben Sie eine nicht-rekursive Methode zur Berechnung der Fakultät einer Zahl.
Lesen Sie die Zahl von der Konsole ein und geben Sie die Fakultät auf der Konsole aus.

Übung

9.4. Fibonacci berechnen

Stellen Sie Ihre Lösung zur Berechnung der Fibonacci Zahl (siehe vorherige Übung) so um, dass die Berechnung eine rekursive Methode verwendet.

Die rekursive Methode soll kein `if` verwenden (aber ggf. ein `switch` **Ausdruck**).

Vergleichen Sie die beiden Lösungen insbesondere für die Berechnung von größeren Fibonacci Zahlen (30, 40, 50, ...).

10. Von Codekonventionen und Lesbarkeit

Lesbarer Java-Code

- Halten Sie sich an die Java-Konventionen.

  (Die Konventionen haben sich - aus guten Gründen - seit Jahrzehnten nicht geändert.)
- Formatieren Sie Ihren Code konsistent; d. h. stellen Sie konsistente Einrückungen sicher!
- Verwenden Sie inhaltsorientierte, sprechende Namen für Variablen, Konstanten, Methoden etc.

※ Hinweis

Manuelles formatieren ist nicht sinnvoll.

Verwenden Sie einen automatische Code-Formatter!

Einrückungen und Blöcke

- Rücken Sie zusammenhängende Blöcke um die gleiche Anzahl von Leerzeichen ein.

  Gängig ist ein Vielfaches von 2 oder 4 Leerzeichen.
- Verwenden Sie keine Tabulatoren (`\t`) für Einrückungen.
- Beginnt ein neuer Block innerhalb eines äußeren Blockes, so werden die zugehörigen Anweisungen tiefer eingerückt als der äußere Block.
- Pro Zeile sollte nur ein Block oder eine Anweisung stehen.

Einrückungen und Blöcke - Beispiele

Falsche Einrückung, fehlende Leerzeichen, fehlende Umbrüche

```
int ggtNaiv(int z1, int z2){
        int min = (z1>z2)?z2:z1; IO.println("current min="+min);
    for(int ggt=min; ggt>1; --ggt){
            if(z1%ggt==0 && z2%ggt==0)
            return ggt;
    }
return 1;
    }
```

Korrekte Einrückung, Leerzeichen und Umbrüche

```
int ggtNaiv(int z1, int z2) {
    int min = (z1 > z2) ? z2 : z1;
    IO.println("current min=" + min);
    for (int ggt = min; ggt > 1; --ggt) {
        if (z1 % ggt == 0 && z2 % ggt == 0)
            return ggt;
    }
    return 1;
}
```

Klammern

- Verwenden Sie Klammern um Blöcke, auch wenn sie nur eine Anweisung enthalten.
  (Insbesondere bei verschachtelten Blöcken bzw. `If`-Anweisungen ist dies wichtig.)
- Bei bedingten Anweisungen und Schleifen steht die öffnende geschweifte Klammer am Ende der 1. Zeile. Die schließende geschweifte Klammer steht in einer eigenen Zeile am Ende. Sie hat die gleiche Einrückung wie die Anweisung.

Warnung

Lange Zeilen, mit mehr als 80 bis 100 Zeichen, erfordern beim Lesen häufig horizontales Scrollen und sind unter allen Umständen zu vermeiden!

Konfigurieren Sie Ihren Editor so, dass Sie unmittelbar sehen, wenn eine Zeile zu lang wird.

90

Methoden und Kommentare

- Dokumentieren Sie Ihre Methoden und Klassen mit Javadoc-Kommentaren.
- Dokumentieren Sie insbesondere die Vor- und Nachbedingungen von Methoden.
- Dokumentieren Sie die Anforderungen an die Parameter.

  Zum Beispiel: `@param n die Zahl für die die Fakultät berechnet wird; n >= 0 und n < 13`
- Dokumentieren Sie die Rückgabewerte.

  Zum Beispiel: `@return die Fakultät von n`
- Der erste Satz eines Javadoc-Kommentar sollte eine kurze, vollständige Beschreibung der Methode enthalten. Dieser wird in der Übersicht verwendet.

  Zum Beispiel: `Berechnet die Fakultät einer Zahl n`
- Dokumentieren Sie keine Trivialitäten

  Zum Beispiel: `i++; // erhöhe i um 1`
- Wenn Sie einen Bedarf sehen, innerhalb einer Methode Kommentare zu schreiben, dann ist dies mgl. ein Hinweis darauf, dass der Code refaktorisiert (*refactored*) werden sollte.

  Zum Beispiel könnten die Methode in kleinere Methoden aufgeteilt werden.
- Dokumentieren Sie insbesondere das, was nicht im Code steht und was nicht offensichtlich ist.

  KI Tools (zum Beispiel GitHub Copilot) sind bereits jetzt in der Lage gute *initiale* Kommentare zu generieren. Aber häufig fehlt die Dokumentation der (impliziten/globalen) Anforderungen sowie der Vor- und Nachbedingungen.

Woher könnte die Anforderung `n < 13` für die Fakultät kommen?

Team und Projektspezifische Konventionen

- die Java-Konventionen sind allgemein gültig und sollten eingehalten werden, decken aber nicht alle Teile des Codes ab.
- In einem Team oder Projekt können spezifische Konventionen festgelegt werden, die über die allgemeinen Konventionen hinausgehen. Diese sollten dann **automatisiert überprüft und ggf. automatisch formatiert werden**.
- Beispiele sind:
  - Einrückungen der Parameter bei Methoden mit „vielen“ Parametern.
  - Sprache in der Variablen benannt werden. (z. B. Fachsprache in Englisch oder Deutsch)
  - Maximale Einrückungstiefe von Schleifen und Bedingungen.
  - ...

Übung

1. Überprüfen Sie den von Ihnen geschrieben Code auf korrekte Formatierung.
2. Installieren Sie für VS Code das Java Extension Pack (falls noch nicht geschehen) und verwenden Sie den eingebauten Code Formatter über die entsprechende Tastenkombination.
   (Auf Mac mit Standardeinstellungen zum Beispiel: `Shift` + `Alt` + `F`.)
3. Schreiben Sie für die Methoden passende Kommentare im Javadoc-Stil.

Am Ende diskutieren wir Ihren Code/Ihre Kommentare.

Von Codekonventionen und Lesbarkeit - Zusammenfassung

Auf dem Weg zu einem professionellen Programmierer (egal in welcher Sprache) ist es wichtig, neben den Sprachkonstrukten auch die geltenden Konventionen zu erlernen und einzuhalten. Diese sind je nach Sprache meist leicht unterschiedlich, aber in der Regel sehr ähnlich.

Das Einhalten fördert die Zusammenarbeit mit anderen Programmierern - *insbesondere auch Ihrem zukünftigen Ich* - und erhöht die Lesbarkeit des Codes.

Übung

10.1. Tage seit Geburt berechnen

Schreiben Sie ein Programm, dass berechnet wie viele Tage ein Mensch bereits auf der Welt ist. Als Eingaben sollen das Geburtsdatum und das aktuelle Datum eingegeben werden. D. h. Sie fragen erst den Tag, dann den Monat und dann das Jahr der Geburt ab. Anschließend fragen Sie den aktuellen Tag, Monat und das Jahr ab. Geben Sie dann die Anzahl der Tage aus.

Verwenden Sie Methoden für sinnvolle Teilaufgaben. (Z. B. Ist ein Jahr ein Schaltjahr, Tage eines Jahres. Tag im Jahr (siehe vorhergehende Übung)).

Anforderungen

- Kommentieren Sie Ihre Methoden sinnvoll.
- Testen Sie Ihren Code mit verschiedenen Eingaben.
- Wenn Sie einen Fehler in der Eingabe finden, geben Sie eine Meldung aus und beenden Ihr Programm mit `System.exit(1)`.

Berechnung eines Schaltjahres:

Ein normales Jahr aus 365 Tagen. Da die Zeit, die die Erde benötigt, um sich einmal um die Sonne zu drehen jedoch 365,2425 Tage beträgt, wird alle vier Jahre ein „Schaltjahr“ von 366 Tagen verwendet, um den durch drei normale (aber kurze) Jahre verursachten Fehler zu beseitigen. Jedes Jahr, das gleichmäßig durch 4 teilbar ist somit ein Schaltjahr: 1988, 1992 und 1996 sind beispielsweise Schaltjahre.

Um den kleinen entstehenden Fehler zu korrigieren, ist ein Jahr, das durch 100 teilbar ist (z. B. 1900 und 2000), nur dann ein Schaltjahr ist, wenn es auch durch 400 teilbar ist (zuletzt z. B. 2000).

11. Assertions

Java Assertions

Assertions sind eine Möglichkeit, um sicherzustellen, dass bestimmte Bedingungen erfüllt sind.

Syntax: `assert <Bedingung>;`

bzw.

Syntax: `assert <Bedingung>: <Ausdruck>;`

Die Bedingung muss ein boolescher Ausdruck sein. Der Ausdruck ist optional und wird nur ausgewertet, wenn die Bedingung falsch ist. Normalerweise wird der Ausdruck verwendet, um eine Fehlermeldung zu erzeugen.

Beispiel: Funktion mit Assertion

```
/// Berechnet den GGT
///
/// @param z1 die erste Zahl; `z1 ⩾ 0`
/// @param z2 die zweite Zahl; `z2 ⩾ 0`
/// @return den GGT von z1 und z2
int ggt(int z1, int z2) {
    assert z1 ⩾ 0 && z2 ⩾ 0 : "z1 und z2 müssen ⩾ 0 sein";
    if (z2 == 0)
        return z1;
    else
        return ggt(z2, z1 % z2);
}
```

```
ggt(-2,4) // ⟹  Exception java.lang.AssertionError: z1 und z2 müssen ⩾ 0 sein
```

Assertions sind gut geeignet zur Überprüfung von:

- (Internen) Invarianten
- Kontrollfluss-Varianten
- Vorbedingungen, Nachbedingungen

⚠ Warnung

Die Auswertung der Bedingung sollte keine Seiteneffekte haben, da diese nur bei aktivierten Assertions überhaupt ausgeführt wird und dies auch *die Erwartungen anderer Programmierer verletzen würde*.

Beispiel: Assertion mit Seiteneffekt

```
int ggt(int z1, int z2) {
    // ⚠ Seiteneffekt, der den Fehler (sogar) korrigiert...
    assert (z1 = Math.abs(z1)) ⩾ 0 && (z2 = Math.abs(z2)) ⩾ 0;

    if (z2 == 0) return z1;
    else         return ggt(z2, z1 % z2);
}
```

※ Hinweis

Java Assertions sollten nur für Bedingungen verwendet werden, die niemals falsch sein dürfen.

Assertions dienen der Identifikation von Programmierfehlern und sollten nicht für Bedingungen verwendet werden, die auf zu erwartende Fehler zurückzuführen sind. (Z. B. falsche Nutzereingaben oder Netzwerkfehler etc.)

- Assertions werden in Java *nur bei expliziter Aktivierung überprüft*.
- Um im Code zu prüfen, ob Assertions aktiviert sind, kann folgender (auf einem Seiteneffekt basierender) Code verwendet werden:

```
1 var assertionsEnabled = false;
2 assert (assertionsEnabled = true);
3 if (assertionsEnabled) { ... }
```

- Um Assertions zu aktivieren, müssen Sie den Kommandozeilenparameter `-ea` oder `-enableassertions` verwenden. Bzw. bei der JShell `-R -ea`.

  Zum Beispiel können Sie die JShell wie folgt starten:

  ```
  jshell -R -ea
  ```

Die Tatsache, dass Assertions nur bei expliziter Aktivierung überprüft werden, ist einer der größten Kritikpunkte an Java Assertions.

Übung

11.1. Assertions

Erweitern Sie Ihre Methode zur Berechnung der Fakultät um Assertions, die sicherstellen, dass die Eingabe nicht negativ und nicht größer als 20 ist.