

Einführung in die Programmierung mit Java

Dozent: Prof. Dr. Michael Eichberg

Kontakt: michael.eichberg@dhbw-mannheim.de, Raum 149B

Version: 1.0

Die Folien sind teilweise inspiriert von oder basierend auf Lehrmaterial von Prof. Dr. Michael Matt.



1

Folien: <https://delors.github.io/prog-java-basics/folien.de.rst.html>

<https://delors.github.io/prog-java-basics/folien.de.rst.html.pdf>

Fehler melden:

<https://github.com/Delors/delors.github.io/issues>

1. EINFÜHRUNG

"Hello World" - das erste Java-Programm

HelloWorld.java^[1]

```
void main(){  
    println("Hello World!");  
}
```

[1] Die Datei HelloWorld.java kann [hier](#) heruntergeladen werden und mit `java --enable-preview HelloWorld.java` ausgeführt werden.

3

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 die Einstiegsmethode in das Programm. Der Text **Hello World!** wird mit der Methode `println` auf der Konsole ausgegeben. Die Methoden `print`, und `println` sind in Java Skripten immer verfügbar und geben den übergebenen Text auf der Konsole aus (ohne bzw. mit Zeilenumbruch am Ende).

Von der Konsole lesen

HelloYou.java^[2]

```
void main() {  
    println("Hello "+ readln("What is your name? "));  
}
```

^[2] HelloYou.java

4

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.

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!

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!

2. EINFACHE PROZEDURALE PROGRAMMIERUNG MIT VARIABLEN, KONSTANTEN, LITERALLEN 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), Wahrheitswerte (boolean)

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

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.

1

Beispiel (ab Java 1.0 - spezifische Tags und HTML)

```
/**
 * 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);
}
```

2

Beispiel (ab Java 23 - spezifische Tags und Markdown)

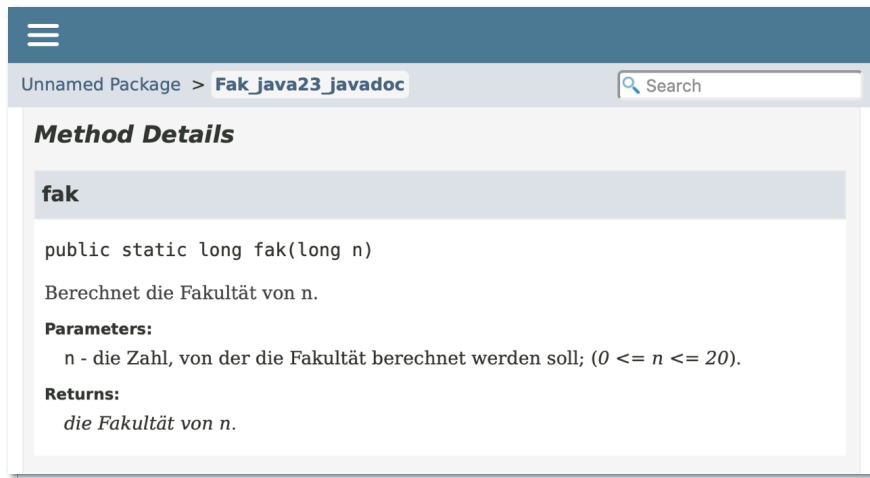
```
/// Berechnet die Fakultät von n.
///
/// @param n die Zahl, von der die Fakultät berechnet werden soll;
///         (*0 <= n <= 20*).
/// @return _die Fakultät von n_.
long fak(long n){ // TODO mögliche Fehlerfälle abfangen
    /* Die Verwendung von long als Datentyp limitiert uns auf n <= 20;
       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);
}
```

3

Erzeugte Dokumentation (mit Java 23)



4

8

JavaDoc tags

@param <name descr>:

Dokumentiert einen Parameter einer Methode.

@return <descr>:

Dokumentiert den Rückgabewert einer Methode.

3. PRIMITIVE DATENTYPEN

Verwendung von Datentypen

- Um die erlaubten Werte von Parametern, Variablen und Rückgabewerten genauer spezifizieren zu können, werden Datentypen verwendet.
- Java stellt hierzu primitive Datentypen, Aufzählungen (**enum**), Klassen und Interfaces 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:

10

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.

1

Umrechnung Binär-Dezimal

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

2

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	-922337022036854775808 bis 922337022036854775807	2^{64}

3

11

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)

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

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

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

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

1

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}

2

12

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 **chars**.

- Für Zeichenketten (z. B. "Hello World") existiert ein nicht-primitiver Datentyp **String**.

14

Unicode Zeichen und chars

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 **chars** und entspricht somit nicht notwendigerweise der Anzahl der (sichtbaren) Zeichen.

```
1 jshell> var smiley = Character.toChars(0x1F60E)
2 smiley ==> char[2] { '?', '?' }
3
4 jshell> var s = new String(smiley)
5 s ==> "😊"
6
7 jshell> s.length()
8 $1 ==> 2
9
10 jshell> s.getBytes(StandardCharsets.UTF_8)
11 $2 ==> byte[4] { -16, -97, -104, -114 }
12
13 jshell> s.codePointCount(0,s.length())
14 $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

16

- 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 (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

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

18

Textblöcke werden seit Java 15 unterstützt.

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

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

1

Beispieldeklaration und -initialisierung

```
void main() {  
    // Deklaration (Datentyp muss konkret angegeben werden)  
    int alter;  
    // Deklaration und Initialisierung inkl. Datentyp (Standardfall)  
    String name = "Asta Mueller";  
  
    // vereinfachte Deklaration mittels var und Initialisierung (seit Java 10)  
    // (Datentyp wird automatisch erkannt)  
    var geburtsOrt = "Berlin";  
    var wohnort = "Schönau";  
    var geschlecht = 'd';  
  
    alter = 25; // späte Initialisierung  
    println(name + "(" + geschlecht + ")", " + alter + " Jahre, aus " + wohnort);  
}
```

2

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

1

Beispieldeklaration und -initialisierung

```
void main() {  
    // Deklaration und Initialisierung inkl. Datentyp (Standardfall)  
    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';  
  
    println(NAME + "(" + GESCHLECHT + ")", " + " Jahre, aus " + WOHNORT);  
  
    // name = "Berta"; // error: cannot assign a value to final variable name  
}
```

2

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`)

23

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.

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.

1

```
# jshell --enable-preview

| Welcome to JShell -- Version 23
| For an introduction type: /help intro

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

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

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

2

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

Bezeichner

```
1 var 1a = ...  
2 var 1_a = ...  
3 var _1a = ...  
4 var a1 = ...
```

```
5 int i;  
6 int _i;  
7 float $$f;  
8 final float E = ...;
```

```
9 String Wohnort;  
10 String ortDerGeburt;  
11 void BucheFlug() {...}  
12 class FlugBuchungen{...}
```

Bezeichner

```
1 var 1a = ...  
2 var 1_a = ...  
3 var _1a = ...  
4 var a1 = ...
```

```
5 int i;  
6 int _i;  
7 float $$f;  
8 final float E = ...;
```

```
9 String Wohnort;  
10 String ortDerGeburt;  
11 void BucheFlug() {...}  
12 class FlugBuchungen {...}
```

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(){ <IHRE CODE> }`).

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 ausgedrückt 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).

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 ausgedrückt 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).

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 = readln("Enter your age: ");  
    int age = Integer.parseInt(s);  
  
    if (age >= 18) {  
        println("You are an adult.");  
    } else {  
        println("You are a minor.");  
    }  
  
    var yearsUntil100 = 100-age;  
    println("You will be 100 in " + yearsUntil100 + " years.");  
}
```

Operatoren und Operanden in der Mathematik

Binäre/Zweistellige Operatoren (🚩 *Binary Operators*)

Addition

1. Operand	Operator	2. Operand
1	+	2

Unäre/Einstellige Operatoren (🚩 *Unary Operators*)

Negation

Operator	Operand
—	(2)

Fakultät

Operator	Operand
2	!

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 bestimmte 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
```

Hinweis

Wenn Sie die folgenden Codeschnipsel ( *Snippets*) in der Java Shell (`jshell`) ausführen möchten, dann müssen sie noch die Methoden `println` und `readln` definieren: `void println(Object o) { System.out.println(o); }` und `String readln(String s) { return System.console().readln(s); }`.

Alternativ können Sie den unten verlinkten Code direkt in die JShell laden:

```
jshell --enable-preview <DATEINAME>
```

Alternative können Sie ein Java Script schreiben (inkl. `main` Methode). In diesem Fall sind die beiden Methoden direkt verfügbar und müssen nicht extra deklariert werden.

Ich empfehle Ihnen, die Beispiele händisch einzugeben, dann lernen Sie mehr!

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)

1

JShell-Beispiel: [ArithmetischeOperatoren.jshell.java](#)

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

Zweistellige Operatoren - welche Werte werden ausgegeben?

2

34

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

Zweistellige Operatoren - welche Werte werden ausgegeben?

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 $\{x_{neu} = x_{alt} - 1; x_{alt}\}$

1

JShell-Beispiel: [ArithmetischeOperatoren.jshell.java](#)

```
1 // Einstellige Operatoren
2 int a = 5;
3 println( ++a );
4 println( a++ );
5 println( -a );
```

Einstellige Operatoren - welche Werte werden ausgegeben?

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

1

JShell-Beispiel: [Vergleichsoperatoren.jshell.java](#)


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

Einstellige Operatoren - welche Werte werden ausgegeben?

2

Einstellige 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  <i>exclusive OR</i>)

1

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

2

JShell-Beispiel: [LogischeOperatoren.jshell.java](#)

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

Logische Operatoren - welche Werte werden ausgegeben?

3

- 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.

Logische Operatoren - welche Werte werden ausgegeben?

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 eingelesenen String direkt an die Methode übergeben oder vorher in einer Variable speichern.

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

```
void main() {  
    // Ihr Code  
}
```

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

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 eingelesenen String direkt an die Methode übergeben oder vorher in einer Variable speichern.

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

```
void main() {  
    // Ihr Code  
}
```

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

Bedingungsoperator

Der Bedingungsoperator:

<Bedingungsausdruck c > ?

<auszuwertender Ausdruck $a_{(c \text{ wahr})}$ falls c wahr >

:

<auszuwertender Ausdruck $a_{(c \text{ falsch})}$ falls c falsch/unwahr>

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

1

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

Beide Ausdrücke $a_{(c \text{ wahr})}$ und $a_{(c \text{ falsch})}$ 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*.

2

Beispiele

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

Verschachtelung ist möglich aber *nicht* empfehlenswert:

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

3

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`).

1

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

2

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"
```

3

Verschlüsselung mit XOR (`EncrvotionWithXOR.ishell.java`):

```
final var key = new java.util.Random().nextInt();  
Integer.toBinaryString(key); // ==> "1001101011100001111001101010011110"  
  
final var income = 13423;  
Integer.toBinaryString(income); // ==> "110100011011111"  
  
// Verschlüsselung von "income" mit "key" mit Hilfe von XOR:  
final var encryptedIncome = income ^ key;  
Integer.toBinaryString(encryptedIncome); // ==> "1001101011100001111111100100100001"
```

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 Ausdrucks ist der zugewiesene Wert mit dem entsprechenden Datentyp.

1

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
```

2

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

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

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

3

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";  
name ==> "Max"  
  
jshell> String greeting = "Hallo " + name + "!";  
greeting ==> "Hallo Max!"
```

2

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 Ausdrucks 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.

1

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

        float f = b + s;

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

2

```
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)`.

3

```
jshell> short s = Short.MAX_VALUE + Short.MAX_VALUE;
```

```
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 Ausdrucks 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)

1

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
```

2

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.toString(Integer.MIN_VALUE) // "-2147483648"
Integer.toString(Integer.MIN_VALUE - 1) // "-2147483649"
Long.toString(Integer.MIN_VALUE - 1L)
// "-2147483649"
```

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 mehr als 292 Stellen vor dem Komma hat.

[illegible]

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)

Beispiele zur Auswertungsreihenfolge

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 (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`

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 (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`

BMI berechnen

Schreiben Sie ein Java Script, dass den Body-Mass-Index (BMI) berechnet. 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, Normalgewicht oder Übergewicht hat. Falls die Person nicht das Normalgewicht hat, geben Sie auch an, wie viel Gewicht sie bis zum Normalgewicht zunehmen oder abnehmen muss.

Der BMI wird nach folgender Formel berechnet: $BMI = \frac{Gewicht}{Größe^2}$.

Beispielinteraktion:

```
Bitte geben Sie Ihr Gewicht in Kilogramm ein: 80
Bitte geben Sie Ihre Größe in Metern ein: 1.80
Ihr BMI beträgt 24.69
Untergewicht: nein
Normalgewicht: nein
```

49

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

```
void main() {
    // Ihr Code
}
```

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

BMI berechnen

Schreiben Sie ein Java Script, dass den Body-Mass-Index (BMI) berechnet. 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, Normalgewicht oder Übergewicht hat. Falls die Person nicht das Normalgewicht hat, geben Sie auch an, wie viel Gewicht sie bis zum Normalgewicht zunehmen oder abnehmen muss.

Der BMI wird nach folgender Formel berechnet: $BMI = \frac{Gewicht}{Größe^2}$.

Beispielinteraktion:

```
Bitte geben Sie Ihr Gewicht in Kilogramm ein: 80
Bitte geben Sie Ihre Größe in Metern ein: 1.80
Ihr BMI beträgt 24.69
Untergewicht: nein
Normalgewicht: nein
Übergewicht: 5.897499999999994 kg bis Normalgewicht
```


Umrechnung von Sekunden

Schreiben Sie ein Java Script, 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.

Beispielinteraktion:

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

Umrechnung von Sekunden

Schreiben Sie ein Java Script, 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.

Beispielinteraktion:

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

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.

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

- 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.

```
void main() {  
    { // Block von Anweisungen  
        int a = 1;  
        println("a = "+a);  
    }  
}
```

- 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

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

; // Leere Anweisung

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

// Methodenaufruf
println("age=" + age);
println("gender=" + gender);
/* 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(readln("Wie alt sind Sie?"));
3     boolean adult = false;
4
5     if (age >= 18) { // if-Anweisung
6         adult = true;
7     }
8
9     println("adult=" + adult);
10 }
```

1

```
1 void main() {
2     var age = Integer.parseInt(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 && readln("Geschlecht (m/w/d)?").charAt(0) == 'm')
10             status = 'a';
11     }
12     println("adult=" + adult+ ", status=" + status);
13 }
```

2

57

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     println("Anzahl der Tage in einem Monat");
3
4     var month = Integer.parseInt(readln("Welchen Monate haben wir(1-12)? "));
5
6     int days = 31;
7     if (month == 2 && readln("Schaltjahr (j/n)? ").charAt(0) == 'j')
8         days = 29;
9     else if (month == 2)
10        days = 28;
11    else if (month == 4)
12        days = 30;
13    // ...
14    println("days=" + days);
15 }
```

1

```
1 void main() {
2     println("Anzahl der Tage in einem Monat");
3
4     var month = Integer.parseInt(readln("Welchen Monate haben wir(1-12)? "));
5
6     int days = 31;
7     if (month == 2 && readln("Schaltjahr (j/n)? ").charAt(0) == 'j')
8         days = 29;
9     else // "nur" umformatiert
10        if (month == 2)
11            days = 28;
12        else
13            if (month == 4)
14                days = 30;
15        // else ...
16    println("days=" + days);
17 }
```

2

58

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.

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>

1

case L :

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

Ein **case**-Block setzt sich zusammen aus dem Schlüsselwort **case**, einem oder mehreren **Literals** (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.

2

```
1 void main() {
2     var month = Integer.parseInt(readln("Welchen Monate haben wir(1-12)? "));
3     int days = 31;
4     switch (month) { // vor Java 14!
5         case 2:
6             if (readln("Schaltjahr (j/n)? ").charAt(0) == 'j')
7                 days = 29;
8             else
9                 days = 28;
10            break;
11        case 4:
12        case 6: case 9: case 11: // <= possible, but unusual formatting
```

```

13         days = 30;
14         break;
15     }
16     println("Anzahl der Tage im Monat " + days);
17 }

```

3

```

1 void main() {
2     var month = Integer.parseInt(readln("Welchen Monate haben wir(1-12)? "));
3     int days = 31;
4     switch (month) { // seit Java 14
5         case 2:
6             if (readln("Schaltjahr (j/n)? ").charAt(0) == 'j')
7                 days = 29;
8             else
9                 days = 28;
10            break;
11        case 4, 6, 9, 11:
12            days = 30;
13            break;
14    }
15    println("Anzahl der Tage im Monat " + days);
16 }

```

4

```

1 void main() {
2     var month = Integer.parseInt(readln("Welchen Monate haben wir(1-12)? "));
3     // seit Java 14:
4     int days = switch (month) { // Switch-Ausdruck
5         case 2:
6             yield readln("Schaltjahr (j/n)? ").charAt(0) == 'j' ? 29 : 28;
7         case 4, 6, 9, 11:
8             yield 30;
9         default:
10            yield 31;
11    };
12    println("Anzahl der Tage im Monat " + days);
13 }

```

5

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-Effekt*, d. h. ein **break** ist nicht zur Beendigung eines **case**-Blocks zu verwenden!

6

```
1 void main() {
2     var month = Integer.parseInt(readln("Welchen Monate haben wir(1-12)? "));
3     int days = 31;
4     switch (month) { // seit Java 14
5         case 2 -> { // Block oder Ausdruck!
6             if (readln("Schaltjahr (j/n)? ").charAt(0) == 'j')
7                 days = 29;
8             else
9                 days = 28;
10        }
11        case 4, 6, 9, 11 ->
12            days = 30;
13    }
14    println("Anzahl der Tage im Monat " + days);
15 }
```

7

```
1 void main() {
2     var month = Integer.parseInt(readln("Welchen Monate haben wir(1-12)? "));
3     // seit Java 14:
4     int days = switch (month) { // Switch-Ausdruck
5         case 2 -> readln("Schaltjahr (j/n)? ").charAt(0) == 'j' ? 29 : 28;
6         case 4, 6, 9, 11 -> 30;
7         default -> 31;
8     };
9     println("Anzahl der Tage im Monat " + days);
10 }
```

8

59

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>.

1

```
1 void main() {
2     var name = readln("Wie ist Dein Name? ");
3     String nameAnalysis = switch (name) {
4         // Der erste Vergleich, der zutrifft, wird ausgeführt.
5         case "Michael", "Tom", "Erik"           -> "m";
6         case "Alice", "Eva", "Maria", "Eva-Maria" -> "w";
7         case String s when s.length() < 2        -> "kein Name";
8         case _ when name.contains("-")           -> "Doppelname";
9         default                                   -> "<unbekannt>";
10    };
11    println("Namensanalyse = " + nameAnalysis);
12 }
```

Erfolgreicher Musterabgleich?

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

2

Wir werden Pattern Matching später detailliert behandeln.

Erfolgreicher Musterabgleich?

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

Effizienz von bedingten Anweisungen

- Bei **if**-/else-Anweisungen werden die Prüf-Ausdrücke sequentiell (in der angegebenen Reihenfolge) ausgewertet (ein Ausdruck pro Alternative).
- Bei **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.