Java-Quelltext übersetzen und ausführen

```
# Projektverzeichnis erstellen.
mkdir projekt
# In Projektverzeichnis wechseln.
cd projekt
# Quelltextdatei erzeugen und editieren. (SPEICHERN!)
notepad MyClass.java
# Quelltextdatei mit Java Compiler übersetzen.
# Erfolgt keine Ausgabe, war die Übersetzung erfolgreich.
javac MyClass.java
# Mit dem Java Launcher eine Klasse starten (deren main-Methode).
java MyClass
```

Hinweis: Kleinere Programme können mit mit dem Java Launcher direkt übersetzt und sofort danach ausgeführt werden. Allerdings wird die class Datei nicht gespeichert. Beispiel:

```
# Launcher erkennt, dass es sich um eine Quelltextdatei handelt.
# Er übersetzt zuerst die Datei und führt sie gleich danach aus.
java MyApp.java
```

Wozu brauchen wir Datentypen?

- Ein Datentyp definiert, wie ein Bitmuster im Speicher zu interpretieren ist. Beispiel: Was bedeutet die Bitfolge `011001`?
- Ein Datentyp legt fest, wie viel Speicher für eine Variable zu reservieren ist. Beispiel: Eine Variable vom Typ `int` belegt 32 Bits (also 4 Bytes) im Speicher.
- Ein Datentyp definiert den Wertebereich für eine Variable. Beispiel: Der Datentyp `byte` lässt nur Werte im Bereich `-128` bis `+127` zu.
- Ein Datentyp legt fest, welche Operationen mit einer Variablen zulässig sind. Beispiel: Einen `int` kann man multiplizieren, aber einen `String` hingegen nicht.
- Ein Datentyp liefert dem Compiler zusätzliche Informationen, damit er die typkonforme Verwendung der Variablen prüfen kann.
- Datentypen legen die Intention für Variablen fest und fördern damit die Verständlichkeit des Quelltextes.

In Java gibt es zwei Kategorien von Datentypen:

- Primitive Datentypen
- Referenzdatentypen

Hinweis: Für jeden primitiven Datentyp existiert in Java ein korrespondierender Referenzdatentyp - sogenannte Wrapper-Klassen. Beispiel: `byte` und `Byte`, `char` und `Character`, `double` und `Double`.

Um den Wertebereich eines primitiven Datentyps zu ermitteln, verwende dessen zugehörige Wrapper-Klasse. Beispiel:

```
Byte.MIN_VALUE
Byte.MAX_VALUE
Integer.MAX_VALUE
Integer.MIN_VALUE
```

Rundungsfehler bei Datentyp `double` und `float`

Mit den Datentypen `double` und `float` können wir Zahlen mit Nachkommastellen abspeichern. Hier kann es jedoch zu Rundungsfehlern kommen. Für `double` gilt: Ungefähr 15 signifikante Ziffern können exakt dargestellt werden. Bei Datentyp `float` sind es hingegen nur etwa 7.

Die *betragsmäßig* größte Zahl ist bei `double` etwa 1.8E308 und die betragsmäßig kleinste Zahl ist 4.9E-328. (E-328 bedeutet "10 hoch -328").

Achtung: Manche Dezimalzahlen, z.B. 0.1, sind im Binärsystem nicht exakt darstellbar. Diese können nur gerundet abgespeichert werden.

Faustregeln bei Typkonvertierungen

Merke:

- Ganze Zahlen haben den Datentyp `int`.
- Zahlen mit Nachkommastellen bzw. Dezimaltrenner (`.`) haben den Datentyp `double`.
- Verwendet man die wissenschaftliche Notation (z.B. `2e-3`) so hat dieser Wert den Datentyp `double`.
- Um eine Gleitkommazahl als Float anzugeben, verwende den Suffix `f`. Beispiel: `2.5f`.

Regeln:

- Wenn man zwei `int` Werte miteinander verrechnet, entsteht wieder ein Ergebnis vom Typ `int`. Nachkommastellen werden abgeschnitten.
- Verrechnet man zwei Werte miteinander, wobei mindestens ein Wert ein `double` ist, dann ist das Gesamtergebnis vom Typ `double`.
- Verrechnet man zwei `byte` Werte miteinander, ist das Ergebnis vom Typ `int`.
- Verrechnet man zwei `short` Werte miteinander, ist das Ergebnis vom Typ `int`.
- Verrechnet man zwei `float` Werte miteinander, ist das Ergebnis vom Typ `float`.

Zahlen angeben in verschiedenen Zahlensystemen

In Java können Zahlen im Binär-, Oktal-, Dezimal- und Hexadezimalsystem angegeben werden.

Binäre Zahlen beginnen mit Präfix `0b`, oktale Zahlen beginnen mit Präfix `0` und hexadezimale Zahlen beginnen mit Präfix `0x`. Ohne Präfix werden Zahlen als Dezimalzahlen interpretiert.

Was war noch mal der Unterschied zwischen Instanzmethoden und statischen Methoden?

Eine Instanz ist ein Objekt einer Klasse. Jede Instanz hat eine Identität, einen Zustand und ein Verhalten.

Instanzmethoden sind Methoden, die auf einem Objekt aufgerufen werden müssen. Diese Methoden haben Zugriff auf die Instanzzustand.

Statische Methoden sind Methoden, die direkt auf der Klasse aufgerufen werden. Sie benötigen kein Objekt, um ihre Aufgabe durchzuführen. Selbst wenn man auf einem Objekt eine statische Methode aufruft, so hat sie dennoch keinen Zugriff auf den Objektzustand. Beispiel: Die Methoden `sin`, `cos`, `pow` der Klasse `Math`. Statische Methoden werden i.d.R. für Algorithmen verwendet oder einfach nur als Hilfsmethoden. Weitere Beispiele: Die Methoden `valueOf` und `format` der Klasse `String`.

In Java werden statische Methoden häufig auch als Factory-Methoden verwendet. Eine Factory-Methode hat die Aufgabe, Objekte einer Klasse zu erzeugen.

Wichtige String-Methoden

Merke: Ein String-Objekt ist unveränderlich. Jeder Methodenaufruf auf einem String-Objekt liefert ein neues String-Objekt als Ergebnis!

Achtung: Bei den meisten Methoden wird die Groß- und Kleinschreibung berücksichtigt!

Merke: Wenn du Zeichenketten vergleichst, verwende statt `==` die `equals` bzw. die `equalsIgnoreCase` Methode! Der Vergleichsoperator `==` prüft, ob zwei Variablen auf *dasselbe* Objekt verweisen, während `equals` prüft, ob die beiden Variablen *gleiche* Werte enthalten.

```
String name = "alice";
name.length(); // 5
name.charAt(0); // 'a'
name.charAt(name.length() - 1); // 'e'
String message = "The result is: " + 123.5; // "The result is 123.5"
String.valueOf(true); // "true"
String.valueOf(456); // "456"
name = "Alice Wonderland";
name.substring(6); // "Wonderland"
name.substring(6, 9); // "Won"
name.toLowerCase(); // "alice wonderland"
name.toUpperCase(); // "ALICE WONDERLAND"
"abc".repeat(3); // "abcabcabc"
" some text  ".trim() // "some text"
"His name is \"Bob\"!"; // His name is "Bob"!
"2024-03-02".split("-"); // String[3] { "2024", "03", "02" }
"2024.03.02".split("\\."); // String[3] { "2024", "03", "02" }
String.join("-", "2024", "03", "02"); // "2024-03-02"
"Alice Bob Charlie".replace(" ", "-"); // "Alice-Bob-Charlie"
"Alice Bob Charlie".replace(" ", ""); // "AliceBobCharlie"
```

```
"Alice".startsWith("Al"); // true
"Alice".startsWith("al"); // false
"Alice".endsWith("ce"); // true
"Alice".endsWith("e"); // true
"Alice".endsWith(""); // true
"Alice".indexOf("ic"); // 2
"Anna".indexOf("n"); // 1
"Anna".lastIndexOf("n"); // 2
// Ergebnis der folgenden Anweisung: "3,00 + 7,000 ergibt A"
"%.2f + %.3f ergibt %X".formatted(3.0, 7.0, 10);
"ABC".equals("abc") // false
"ABC".equalsIgnoreCase("abc"); // true
"Alice Wonderland".contains("Won"); // true
"abc".compareTo("abd"); // -1, d.h. "abc" < "abd"
"abd".compareTo("abc"); // 1, d.h. "abd" > "abc"
"abc".compareTo("abc"); // 0, d.h. "abc" = "abc"
```

Bearbeitbare Zeichenketten mit dem StringBuilder

Mit der Klasse StringBuilder können wir Zeichenpuffer erstellen und diese Zeichen direkt bearbeiten, ohne jedes Mal einen neuen String zu erzeugen. Das ist wesentlich effizienter und ressourcenschonender.

Merke: Methoden verändern das StringBuilder-Objekt selbst. Um ein String-Builder Objekt in einen String zu konvertieren, rufe Methode `toString` auf.

Hinweis: Viele Methoden, die in der Klasse String vorhanden sind, gibt es auch im StringBuilder.

```
String lastName = "Wonderland";
StringBuilder buffer = new StringBuilder(lastName);
buffer.insert(0, "Alice "); // "Alice Wonderland"
buffer.setCharAt(0, 'a'); // "alice Wonderland"
buffer.delete(0, 6); // "Wonderland"
buffer.append(" from Oz"); // "Wonderland from Oz"
String newLastName = buffer.toString();
```

Homogene Daten speichern mit Arrays

Ein Array ist eine lineare Datenstruktur, die ihre Elemente sequenziell anordnet. Jedes Element hat eine feste Position (Index). Das erste Element hat den Index 0. Die Elemente müssen denselben Datentyp haben.

Merke: Ein Array kann strukturell nicht verändert werden. D.h. Elemente können weder entfernt noch eingefügt werden. Ein Element lässt sich jedoch mit einem neuen Element ersetzen.

Hinweis: Die Kurzschreibweise `{ e1, e2, ... }` ist nur bei *Variablendefinitionen* zulässig. Will man der Array-Variablen später ein neues Array-Objekt zuweisen, geht dies nur mit dem Operator `new`.

```
int[] data = new int[100]; // int[100] { 0, 0, ..., 0 }
int[] primes = { 2, 3, 5, 7, 11 };
primes.length; // 5
primes[0]; // 2
primes[1]; // 3
primes[0] = 19; // int[5] { 19, 3, 5, 7, 11 }
primes = { 11, 13, 17, 19 }; // Fehler!!!
primes = new int[] { 11, 13, 17, 19 }; // int[4] {...}
```

Die Hilfsklasse `java.util.Arrays` bietet zahlreiche Methoden an, um Array-Objekte zu verarbeiten. Beispiele:

```
int[] numbers = { 6, 2, 7, 1 };
Arrays.sort(numbers); // int[4] { 1, 2, 6, 7}
Arrays.fill(numbers, 2); // int[4] { 2, 2, 2, 2}
int[] someOtherNumbers = { 2, 2, 2, 2 };
Arrays.equals(numbers, someOtherNumbers); // true
someOtherNumbers = new int[] { 2, 2, 2 };
Arrays.equals(numbers, someOtherNumbers); // false
String[] names = { "alice", "bob", "charlie", "damian" };
// Achtung: binarySearch erwartet, dass die Elemente des Arrays
// aufsteigend sortiert sind!
Arrays.binarySearch(names, "bob"); // 1
int[] b1 = { 7, 3, 5 };
int[] b2 = { 7, 3, 5, 6 };
Arrays.compare(b1, b2); // -1, d.h. b1 < b2
Arrays.compare(b2, b1); // 1, d.h. b2 > b1
Arrays.compare(b2, b2); // 0, d.h. b2 gleich b2
int[] copy = b1.clone(); // int[3] { 7, 3, 5 }
numbers = new int[] { 5, 2, 1, 10, 19, 25 };
Arrays.copyOf(numbers, 3); // int[3] { 5, 2, 1 }
Arrays.copyOfRange(numbers, 3, 3 + 2); // int[2] { 10, 19 }
```

Verzweigen mit der if-Anweisung

Hinweis: Die Bedingungen müssen boolesche Ausdrücke sein, d.h. die Berechnungen müssen entweder `true` oder `false` ergeben. Numerische Werte sind nicht erlaubt.

Die Verzweigungen `else if` und `else` sind immer optional.

```
public static void main(String[] args) {
    if (args.length >= 3) {
        System.out.printf(
            "Name mit Anrede: %s %s %s\n",
            args[0], args[1], args[2]);
    } else if (args.length >= 2) {
        System.out.printf(
```

```
                "Name: %s %s\n", args[0], args[1]);
        } else if (args.length >= 1) {
            System.out.printf("Vorname: %s\n", args[0]);
        } else {
            System.out.println(
                "Rufen Sie das Program wie folgt auf:"
                + "[Anrede] Vorname Nachname");
        }
    }
```

Operatoren

Jeder Operator hat eine vordefinierte Priorität und ggf. eine Assoziativität. Die Assoziativität legt fest, ob zwei nebeneinanderstehende Operatoren gleicher Priorität von links oder rechts ausgewertet werden. Beispiel:

```
int age = 17;
boolean isAdult = age >= 18;
// Zwei Operatoren: >= und =
// Operatoren nach Priorität: >=, =
// (boolean isAdult = (age >= 18))

// 4 Operatoren: =, +, *, -
// Operatoren nach Priorität : *, +/-, =
double result = 8 + 3 * 7 - 1
// Auswertung durch Compiler:
// (double result = ((8 + (3 * 7)) - 1))

// 3 Operatoren: =, =, +
// Operatoren nach Priorität: +, =
// Assoziativität von = ist rechts.
int x;
int y;
x = y = 3 + 1
// Auswertung durch Compiler:
// (x = (y = (3 + 1)))
```

Operator	Bedeutung	Ergebnisdatentyp
==	Vergleicht Werte miteinander; Prüft auf Identität	Boolean
!=	Testet auf Ungleichheit bzw. auf unterschiedl. Identität	Boolean
<	Kleiner als	Boolean
>	Größer als	Boolean
<=	Kleiner gleich als	Boolean
>=	Größer gleich als	Boolean

Operator	Bedeutung	Ergebnisdatentyp
&&	Logische UND-Verknüpfung	Boolean
||	Logische ODER-Verknüpfung	Boolean
!	Logische Negation	Boolean
+	Addition oder String-Konkatenation	Numerisch oder String
-	Subtraktion	Numerisch
*	Multiplikation	Numerisch
/	Division (Integerdivision / Gleitkommadivision)	Numerisch
%	Modulo (Division mit Rest)	Numerisch
>>	Bitshift nach rechts	Numerisch
<<	Bitshift nach links	Numerisch
&	Bitweises UND	Numerisch
|	Bitweises ODER	Numerisch
^	Bitweises ENTWEDER-ODER (XOR)	Numerisch
()	Call-Operator (Methoden aufrufen)	variabel
[]	Index-Operator (Elementzugriff bei Arrays)	variabel
.	Member-Access-Operator (Zugriff auf Felder, Methoden)	variabel
=	Zuweisungsoperator	variabel
++	Inkrement-Operator (Addieren von 1)	numerisch
--	Dekrement-Operator (Subtrahieren von 1)	numerisch

Wiederholungen mit der while-Schleife

Hinweis: Die Anweisung `break` innerhalb der while-Schleife beendet die Schleife vorzeitig. Die Anweisung `continue` hingegen, springt direkt zum Kopf der Schleife, damit die Bedingung neu geprüft wird.

```
while (index < args.length) {
    System.out.printf("Argument %d: %s\n", index + 1, args[index]);
    index++;
}
```

Wiederholungen mit der do-while-Schleife

Diese Schleife ist fußgesteuert. Sie wird mindestens einmal ausgeführt. Die `break` Anweisung verlässt die Schleife vorzeitig. Die `continue` Anweisung springt direkt zum "Fuß" der Schleife, also zum Bedingungsausdruck.

```
do {
    System.out.printf("Argument %d: %s\n", index + 1, args[index]);
    index++;
} while (index < args.length); // <- Semikolon!
```

Wiederholungen mit der for-Schleife

Die for-Schleife besteht aus drei Bereichen: Initialisierungsbereich, Bedingungsbereich und "Iterationsbereich".

Der Initialisierungsbereich wird einmal bei Betreten der Schleife ausgeführt. Der Bedingungsbereich wird *vor* jedem Schleifendurchlauf geprüft. Der Iterationsbereich wird *nach* jedem Schleifendurchlauf ausgeführt.

Die Anweisung `break` verlässt die Schleife vorzeitig, ohne dass der Iterationsbereich noch einmal ausgeführt wird. Die Anweisung `continue` springt direkt zum Iterationsbereich und danach zum Bedingungsbereich.

Hinweis: Lässt man den Bedingungsbereich leer, ist das gleichbedeutend mit `true`. Die Schleife wird also unendlich oft ausgeführt - eine sogenannte *Endlosschleife*.

```
for (int i = 0; i < args.length; i++) {
    System.out.printf("Argument %d: %s\n", i + 1, args[i]);
}

System.out.println("\n\n"); // Leerzeilen einfügen.
for (int a = 0, b = 5 ; b >= 0 ; a++, b--) {
    System.out.printf("a = %d und b = %d\n", a, b);
}
```

Wiederholen mit der foreach-Schleife

Die foreach-Schleife vereinfacht die Verarbeitung von sequenziellen Datenmengen. Für jedes Element der Datensequenz wird der Schleifenrumpf ausgeführt. Um das Datenelement anzusprechen, das gerade abgearbeitet wird, muss man eine Laufvariable definieren.

Hinweis: Die foreach-Schleife funktioniert mit allen Datenobjekten, deren Datentyp die Schnittstelle `Iterable` implementiert.

Allgemeine Syntax:

```
for (Datentyp element : datenmenge) {
    // tue etwas mit element
}
```

Hier ein paar konkrete Beispiele:

```
for (String argument : args) {
    System.out.printf("Argument: %s\n", argument);
}

int[] primes = { 2, 3, 5, 7, 11, 13 };
for (int prime : primes) {
    System.out.println(prime);
}

// Klasse String ist kein Iterable. Deshalb muss ein String
// vorher in ein Array konvertiert werden.
for (char c : "Alice".toCharArray()) {
    System.out.printf("%c ", c);
}

// Die Laufvariable kann nicht dazu benutzt werden, Elemente
// im Array zu überschreiben! Die Laufvariable ist lediglich
// eine Kopie des Elements.
int[] numbers = { 1, 2, 3 };
for (int n : numbers) {
    n = n * 2; // Kein Effekt auf numbers!
}
```

Fallunterscheidungen mit der switch-Anweisung/Expression

Mittlerweile unterstützt die `switch` Anweisung beliebige Datentypen. Früher war man auf `int`, `char`, `String` beschränkt. Switch Statements unterstützen neuerdings auch sogenannte Patterns. Das ist ein erweitertes Feature, das wir erst später behandeln.

Hinweis: Verwendet man Zeichenketten (`String`) wird der Vergleich mittels `equals` vorgenommen. Die Groß und Kleinschreibung wird dadurch also berücksichtigt.

Besonders hilfreich bei der modernen switch-Variante ist das automatische Einfügen von `break`. Diese Funktionalität wird aber nur dann bereitgestellt, wenn wir nach dem Schlüsselwort `case` einen Pfeil `->` verwenden, statt eines Doppelpunktes `:`.

Hinweis: Fall-Throughs sind in Java möglich, wenn man das `break` in einem `case` weglässt.

Allgemeine Syntax:

```
// Alte Variante
switch (ausdruck) {
    case konstante1:
```

```
            anweisungen;
            break;
        case konstante2:
            anweisungen;
            break;
        default:
            anweisungen;
            break;
    }

    // Moderne Variante
    switch (ausdruck) {
        // Wird {} verwendet, entfällt das Semikolon. Es können
        // im {} Block mehrere Anweisungen ausgeführt werden.
        case konstante1, konstante2 -> { anweisungen }
        // Eine Anweisung ohne {} Block benötigt ein Semikolon.
        case konstante3 -> anweisung;
        default -> anweisung;
    }

    // In Form einer Expression:
    switch (ausdruck) {
        // Schlüsselwort yield dient zur Rückgabe eines Wertes
        // in einem {} Block.
        case konstante1, konstante2 -> { anweisungen; yield wert; }
        // Angabe von yield hier nicht notwendig, da kein {} Block.
        case konstante3 -> wert;
        default -> wert;
    }
```

Fehler behandeln mit try-catch-finally