

Java für C++-Programmierer

HS 2024, Peter Bühler

Fabian Suter, 12. November 2024

0.0.1



1 Basics

1.1 Grundsätzlich

Diese Zusammenfassung dient zur Hilfe beim Programmieren mit Java anhand von C++-Vorwissen. Es gibt einige wichtige Unterschiede, **Java**:

- kennt keine Zeiger / Pointer
- kennt keine Funktionen (reine OO-Sprache)
- kennt kein Überladen von Operatoren
- kennt keine Destruktoren (Garbage-Collector gibt Speicher frei)
- kennt keine Mehrfachvererbung
- stellt eine umfangreiche Klassenbibliothek zur Verfügung
- -Programme sind plattformunabhängig
- ist weniger hardwarenah wie C++
- ...

1.2 Sourcecode Hello world Java

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

1.3 Java Runtime

Der Compiler erzeugt anders als in C++ **Bytecode**, welcher anschliessend auf einer Java Virtual Machine (JVM) laufen kann. Dadurch wird Java plattformunabhängig, da jede Plattform eine JVM hat, sei es Windows, Android oder MacOS.

2 Datentypen

Ein Datentyp besteht aus *Werten* und *Operationen*. Java besitzt zwei generelle Datentypen, namentlich **Primitive Datentypen** und **Referenzdatentypen**.

2.1 Primitive Datentypen

Sie haben plattformunabhängig den gleichen Wertebereich, es gibt keine **unsigned**-Typen. Für bool'sche Werte gibt es den Datentyp **boolean**.

Typ	Beschr.	Beispiele
boolean	Bool'scher Wert	true, false
char	Textzeichen (UTF16)	'a', 'B', etc.
byte	Ganzzahl (8 Bit)	-128 bis 127
short	Ganzzahl (16 Bit)	-32'768 bis 32'767
int	Ganzzahl (32 Bit)	-2 ³¹ bis 2 ³¹ - 1
long	Ganzzahl (64 Bit)	-2 ⁶³ bis 2 ⁶³ - 1, 1L
float	Gleitkommazahl (32 Bit)	0.1f, 2e4f
double	Gleitkommazahl (64 Bit)	0.1, 2e4

Gleitkommazahlen ohne Angaben sind automatisch **double**.

2.1.1 Überlauf / Unterlauf

Bei **Ganzzahlen** ist der Überlauf in Java definiert, im Gegensatz zu C++.

2147483647 + 1 → -2147483648

Bei **Gleitkommazahlen** gilt dasselbe:

2 * 1e308 → POSITIVE_INFINITY

5e-324 / 2 → 0.0

2.1.2 Undefinierte Operationen

Ganzzahlen werfen bei Division/Modulo durch 0 einen Fehler bzw. eine Exception.

Gleitkommazahlen werfen bei Division durch 0 ein POSITIVE_INFINITY resp. NEGATIVE_INFINITY

Bei undefinierten Rechnungen wie 0 / 0 wird NaN zurückgegeben.

2.1.3 Text-Literale

char mit Apostrophen: 'A', '\n' (NewLine), '\'' (Apostroph)

String mit Anführungszeichen: "Say \"hello\"!\n"

2.2 Referenzdatentypen

2.2.1 Arrays

Arrays speichern wie in C++ mehrere Elemente mit selbem Datentyp. Der Zugriff erfolgt über Index, die Elemente liegen nebeneinander im Speicher. Die Grösse des Arrays muss bei der Deklaration festgelegt werden und ist später nicht mehr änderbar.

Die Anzahl der Elemente kann in Java direkt mit **length** abgerufen werden, siehe auch 4.2.

Falls der Index ungültig ist, wird eine **ArrayIndexOutOfBoundsException** geworfen.

2.3 Typumwandlung

Implizit:
Klein zu Gross,
kein Cast-Operator erf.

```
int a = 4711;
float b = a;
double c = b;
```

Explizit:
Gross zu Klein,
Cast-Operator erf.

```
int y = 0x11223344;
short x = (short) y; // x = 0x3344

double v = 4.7;
int w = (int) w; // w = 4
```

3 Objektorientierung

3.1 Methoden

3.1.1 Parameter

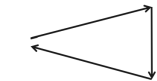
Java verwendet *immer* **Call by Value**. Die Argumente werden kopiert und als Parameter übergeben. Dies kann zu unterschiedlichem Verhalten je nach Datentyp führen:

Primitive Datentypen:

Methode arbeitet mit Kopie des Wertes

```
int x = 5;
square(x);

// x bleibt 5
```



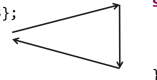
```
static void square(int x) {
    x = x * x;
    System.out.println(x);
}
```

Referenzdatentypen:

Methode arbeitet mit Kopie der Referenz

```
int[] a = {1, 2, 3};
square(a);

// a geändert
```



```
static void square(int[] a) {
    for (int i=0; i<a.length; ++i) {
        a[i] = a[i] * a[i];
    }
}
```

3.2 Unit Testing

Unit Testing ist eine der Varianten, um Bugs zu verhindern. In guten Unit Tests sollen möglichst alle relevanten Fälle abgedeckt sein. Dazu gehören Standardfälle (im Bereich der Funktion) und Edge Cases (z.B. 0, max. und min. Bereich, usw.). Siehe auch 4.4 Faustregel:

- Pro Klasse eine Test-Klasse
- Pro Testfall eine Methode

3.2.1 Assert-Methoden

Methoden	Bedingung
assertEquals(expected, actual)	für prim. und Referenztypen
assertNotEquals(expected, actual)	
assertSame(expected, actual)	actual == expected
assertNotSame(expected, actual)	actual != expected
assertTrue(condition)	condition
assertFalse(condition)	!condition
assertNull(value)	value == null
assertNotNull(value)	value != null

```
package javac.v2.demo;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
```

```
class Demo07AbsTest {

    @Test
    @DisplayName("Positive Number")
    void testPositiveValue() {
        long in = 23;
        long out = 23;
        assertEquals(out, Demo07Abs.abs(in));
    }
}
```

Optionaler Name
für Anzeige

Prüfe, ob Resultat stimmt

4 Code-Snippets

4.1 Fakultät

```
public class Factorial {
    public static void main(String[] args) {
        int n = 12;
        int p = 1;
        int i = 1;
        while (i <= n) {
            p = p * i;
            ++i;
        }
        System.out.println(p);
    }
}
```

4.2 Array-Loop

```
int[] array = {1, 2, 3};

for(int i=0; i < array.length; i++){
    System.out.printf("Index %d : %d\n", i, array[i]);
}

// Index 0 : 1
// Index 1 : 2
// Index 2 : 3

int[][] m = new int[2][3];

// int[Zeile][Spalte]
// m.length    => Anzahl Zeilen
// m[0].length => Anzahl Spalten
```

4.3 Schleife mit continue

4.4 Unit Test