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 null-Referenz

Spezielle Referenz auf "kein Objekt", vordefinierte Konstante, welche für alle Referenztypen gültig ist. *Dereferenzieren* der `null`-Referenz generiert eine `NullPointerException`

2.2.2 Klassen

Siehe 3.1

2.2.3 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 **6.2**.

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

2.2.4 Enums

Ähnliche Verwendung wie in C++. Enums können als Konstantenlisten verwendet werden:

```
public enum Weekday{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
     ;
    public boolean isWeekend(){
        return this == SATURDAY || this == SUNDAY;
    }
}
Weekday currentDay = ...
if (currentDay.isWeekend()){...}
```

2.2.5 Collections

Interface	Klassen	Eigenschaften
List	ArrayList, LinkedList	Folge von Elementen, Duplikate möglich
Set	HashSet, TreeSet	Menge von Elementen, keine Duplikate
Map	HashMap, TreeMap	Abbildung Schlussel → Werte, keine Duplikate

Siehe auch Kapitel 5

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

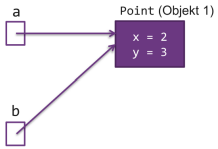
3 Objektorientierung

3.1 Klassen und Objekte

Klassen bestehen aus **Variablen** und **Methoden**.
Objekte können aus Klassen erzeugt werden: `Point a = new Point()`
Konstruktoren können in den meisten IDEs automatisch generiert werden.
Sofern kein Konstruktor definiert ist, wird der **Default-Konstruktor** verwendet. Die Instanzvariablen werden mit Default-Werten initialisiert (primitive mit `0`, Referenzdatentypen mit `NULL`)

Klassen arbeiten mit Referenzen, dies muss bei Zuweisungen beachtet werden:

```java
Point a = new Point();
a.x = 0;
a.y = 1;

Point b = a;
b.x = 2;
b.y = 3;

// Ausgabe: 2, 3
System.out.println( a.x + ", " + a.y);
```



3.2 Methoden

3.2.1 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

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

// x bleibt 5
```

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

Referenzdatentypen:
Methode arbeitet mit Kopie der Referenz

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

// a geändert
```

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

3.3 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 6.4
Faustregel:

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

3.3.1 Assert-Methoden

Methode	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

```java
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

3.3.2 Sichtbarkeit

Keyword	Sichtbar für
public	Alle Klassen
protected	Klassen im selben Package und abg. Klassen
(keines)	Klassen im selben Package
private	Nur eigene Klasse

Für Zugriffe auf `private`-Variablen können Getter- und Setter-Methoden definiert werden. Siehe auch 6.7

4 Variadische Methoden

Falls bei Funktionen die Anzahl der Argumente nicht im Voraus bekannt ist, kann folgende Syntax verwendet werden:
`static int sum(int... numbers){}`
Der Compiler generiert ein Array aus der Parameterliste.

5 Collections

In Collections können nur Referenzdatentypen abgelegt werden. Beim Hinzufügen des Elements wird das Objekt selber **nicht** kopiert, es wird nur eine Referenz abgelegt.

5.1 Wrapper-Objekt

Um primitive Datentypen in Collections verwenden zu können, müssen sie verpackt (*Wrapping*) werden. Dies geschieht meist **implizit**, es muss nur beim Datentyp der Collection definiert werden.

- Für alle primitiven Datentypen gibt es eine Wrapperklasse

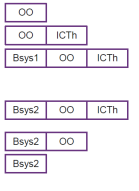
```java
// Boxing
int x = 123;
Integer obj = Integer.valueOf(x);

// Unboxing
Integer obj = ..
int x = obj.intValue();
```

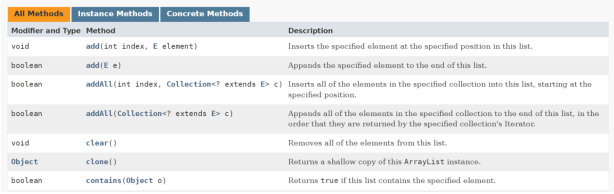
Primitiver Typ	Wrapper-Klasse
char	Character
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

5.2 ArrayList

ArrayLists sind eine geordnete Folge von Elementen mit demselben Referenzdatentyp. Elemente können einfach hinzugefügt oder entfernt werden.
Der Zugriff auf Elemente erfolgt über Index (0 ... size()-1). Die Liste verwendet intern ein Array zur Verwaltung der Elemente. Zu Beginn enthält sie, sofern nicht anders definiert, 10 Elemente und wird bei Erreichen der Kapazität mit Faktor 1.5 mulitpliziert.

```java
ArrayList<String> module = new ArrayList<>();

module.add("OO");           // append at end
module.add("ICTh");
module.add(0, "Bsys1");     // insert at index 0

String x = module.get(1);   // get at index 1
module.set(0, "Bsys2");     // replace at index 0

module.remove("ICTh");      // remove element
module.remove(1);           // remove at index 1

int size = module.size();   // get length list
```





5.3 LinkedList

5.4 HashSet

5.5 TreeSet

5.6 HashMap

5.7 TreeMap

6 Code-Snippets

6.1 Fakultät

```java
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);
    }
}
```

6.2 Array-Loop

```java
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
```

6.3 Schleife mit continue

6.4 Unit Test

¡¡¡¡¡¡¡ HEAD

6.5 Getter- und Settermethoden

6.6 Strings mit ...

=======

6.7 Getter- und Settermethoden

¿¿¿¿¿¿¿ 55516451d11582e5cff10e1155e10ed0ad9fdb32