HOCHSCHULE DER MEDIEN

Java Recap

Software-Entwicklung 2

Bild: Midjourney

Java

- Programme, die in Java geschrieben sind, sind plattformunabhängig: „Write once, run everywhere“ (**WORA**).
- Java-Programme sind einmal geschrieben und können (theoretisch) überall laufen.
- Java-Programme laufen aber nicht direkt auf dem Computer, sondern benötigen eine Zwischenschicht (virtuelle Maschine oder VM): die **Java Runtime**.
- Java Runtime Environments (JREs) gibt es für viele Plattformen: Windows, Mac, Linux, etc.

Eine kleine Geschichte

- Gestartet 1991 von James Gosling, Mike Sheridan und Patrick Naughton als Object Application Kernel (OAK).
- Ursprünglich für interaktives Fernsehen gedacht (zu fortschrittlich damals).
- Java 1.0 wurde 1996 von Sun Microsystems veröffentlicht.
- Der Name stammt von „Java City – roasters of fine coffee“ (ein Café in Menlo Park).
- IDE-Name „JavaBeans“ leitet sich auch von dort ab.Kein Java 2.0, 3.0 oder 4.0 – stattdessen „Java 2 Platform“ (Version 1.2).
- 2004: Java 5.0 (mit den meisten OOP-Features, die wir heute kennen).
- 2006: Veröffentlichung als Open Source (GPL 2.0).
- 2008: Android OS von Google (kein JRE → Android Runtime ART).
- 2009–2010: Oracle kauft Sun Microsystems.
- 2011: Java wird aus dem Linux Standard Base (LSB) entfernt.
- 2020: Java 15 veröffentlicht.

Java vs. C++

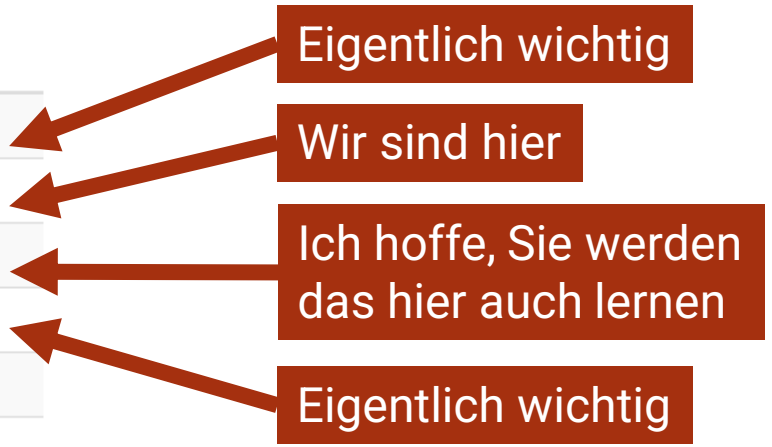
- Java basiert im wesentlichen auf der Syntax von C++.
- Kein Pointer-Arithmetik in Java.
- Klassen können nur eine Elternklasse haben (Single Inheritance).
  - Aber: Interfaces ermöglichen Mehrfachvererbung.
- Java hat Garbage Collection (keine Destruktoren, sondern finalize).
- Keine Operatorüberladung (z. B. + für Vektoren wie in C++).
- C++ Templates ↔ Java Generics (ähnlich, aber nicht gleich!).
- C++ Namespaces ↔ Java Packages (ähnlich, aber nicht gleich!).
- const in C++ entspricht final in Java.
- Parameter-Initialisierung in Java nicht so einfach wie in C++.

Java vs. C#

- C# unterstützt Pointer.
- C# unterstützt unsigned Zahlen (Java nicht).
- C# unterstützt delegates (Referenzen auf Methoden).
- C# unterstützt dynamische Aufrufe (late binding).
- C# unterstützt Referenzen auf Input/Output-Parameter (in Java nur mit Workarounds möglich).
- Methoden in C#: nicht-virtuell standardmäßig (müssen explizit als virtual markiert werden).
- Methoden in Java: immer virtuell (außer static oder private).
- Java unterstützt Phantom References.
- Java unterstützt innere Klassen.

Warum lernen viele Java?

Oct 2020	Oct 2019	Change	Programming Language	Ratings	Change
1	2	↑	C	16.95%	+0.77%
2	1	↓	Java	12.56%	-4.32%
3	3		Python	11.28%	+2.19%
4	4		C++	6.94%	+0.71%
5	5		C#	4.16%	+0.30%
6	6		Visual Basic	3.97%	+0.23%
7	7		JavaScript	2.14%	+0.06%
8	9	↑	PHP	2.09%	+0.18%
9	15	⇈	R	1.99%	+0.73%
10	8	↓	SQL	1.57%	-0.37%
11	19	⇈	Perl	1.43%	+0.40%
12	11	↓	Groovy	1.23%	-0.16%



Warum lernen viele mit Java?

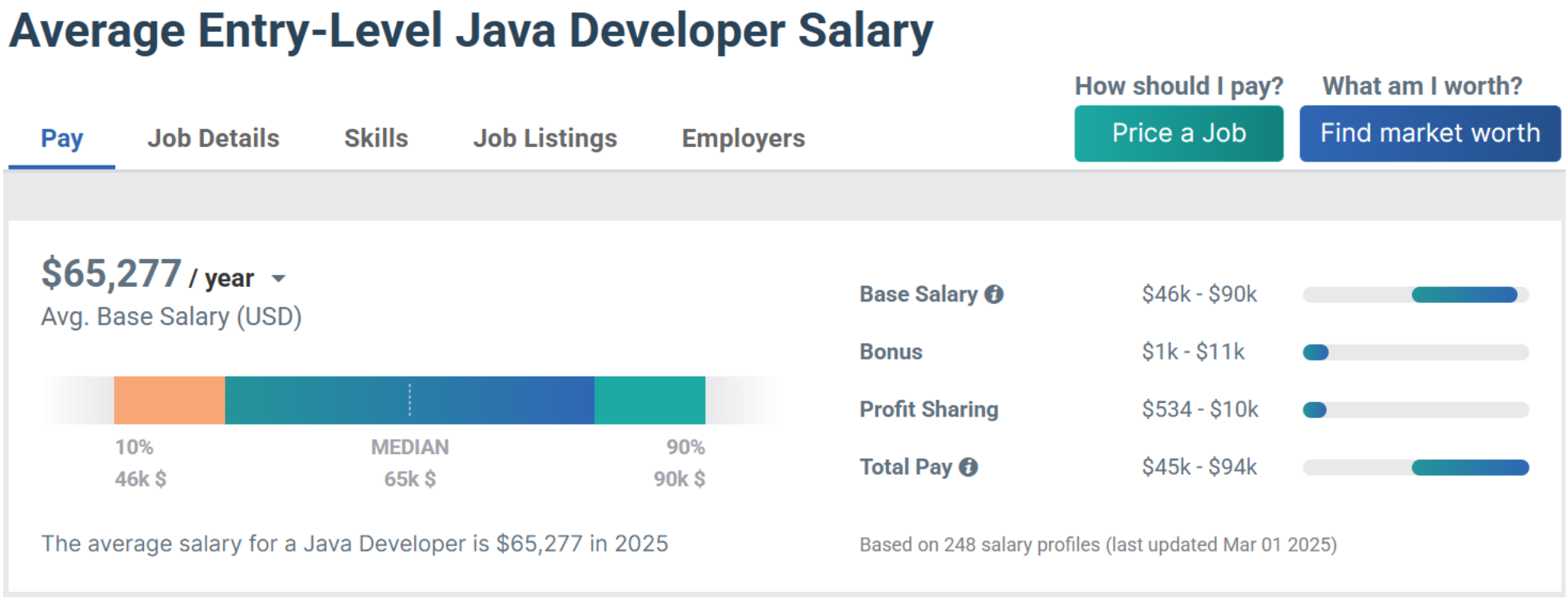
- **Einsteigerfreundlich**: Die Syntax ist relativ klar und verständlich.
- **Verbreitet**: Java gehört seit Jahrzehnten zu den Top-Programmiersprachen weltweit.
- **Vielseitig**: Von Webanwendungen über Apps bis hin zu großen Unternehmenssystemen oder sogar Raumfahrt wird Java eingesetzt.
- **Community & Ressourcen**: Unmengen an Tutorials, Büchern, Forenbeiträgen

  → ideal zum Lernen.
- **Job-Markt**: Java-Entwickler gehören nach wie vor zu den gefragtesten Programmierern.

Für was wird Java verwendet?

- Klassische „Hochsprachen-Programmierung“ (menschlich lesbarer Code).
- Sehr stabil und effiziente Speicherverwaltung.
- Einsatzgebiete:
  - Internet of Things (IoT)
  - APIs, Big Data, Datenbanken
  - E-Commerce-Websites
  - Hochfrequenz-Finanzhandel
  - Wissenschaftliche Anwendungen
  - Mehr als 70 % aller Smartphones weltweit laufen auf Android → die meisten Apps nutzen Java oder Kotlin.
- Läuft u. a. in Gmail und sogar im Curiosity Rover auf dem Mars.
- Von Natur aus multithreaded (kann viele Aufgaben parallel verarbeiten).
- „Javapocalypse“ (YouTube: https://www.youtube.com/watch?v=E3418SeWZfQ).

Verdiene ich irgendwas mit Java?

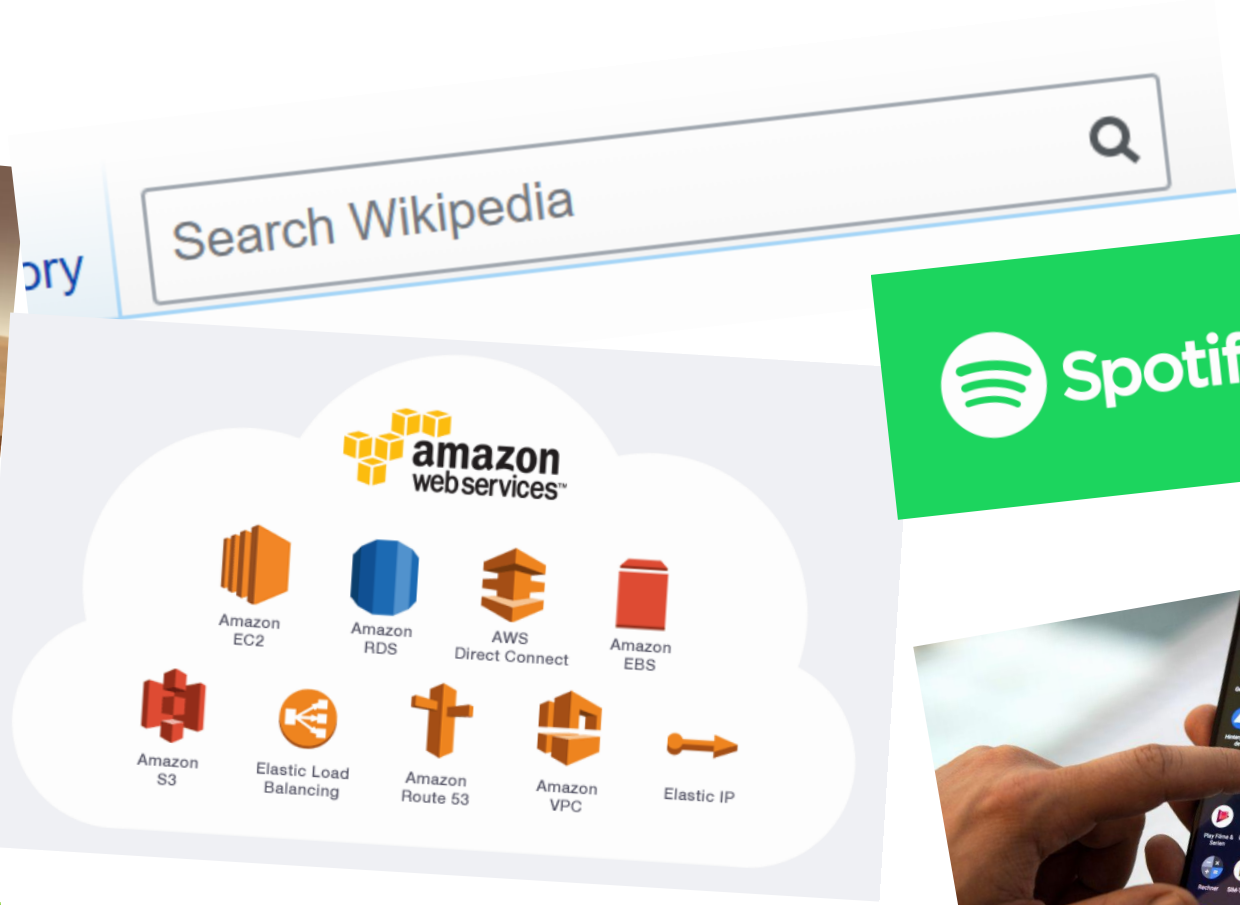


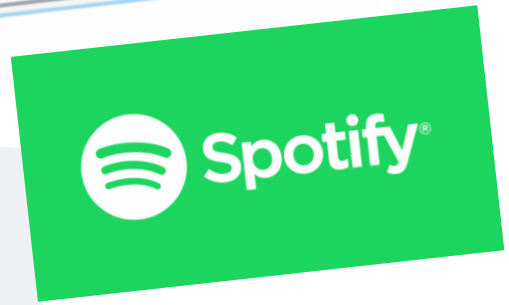
https://www.payscale.com/research/US/Job=Java_Developer/Salary/ea5e82c2/Entry-Level

Warum Java?

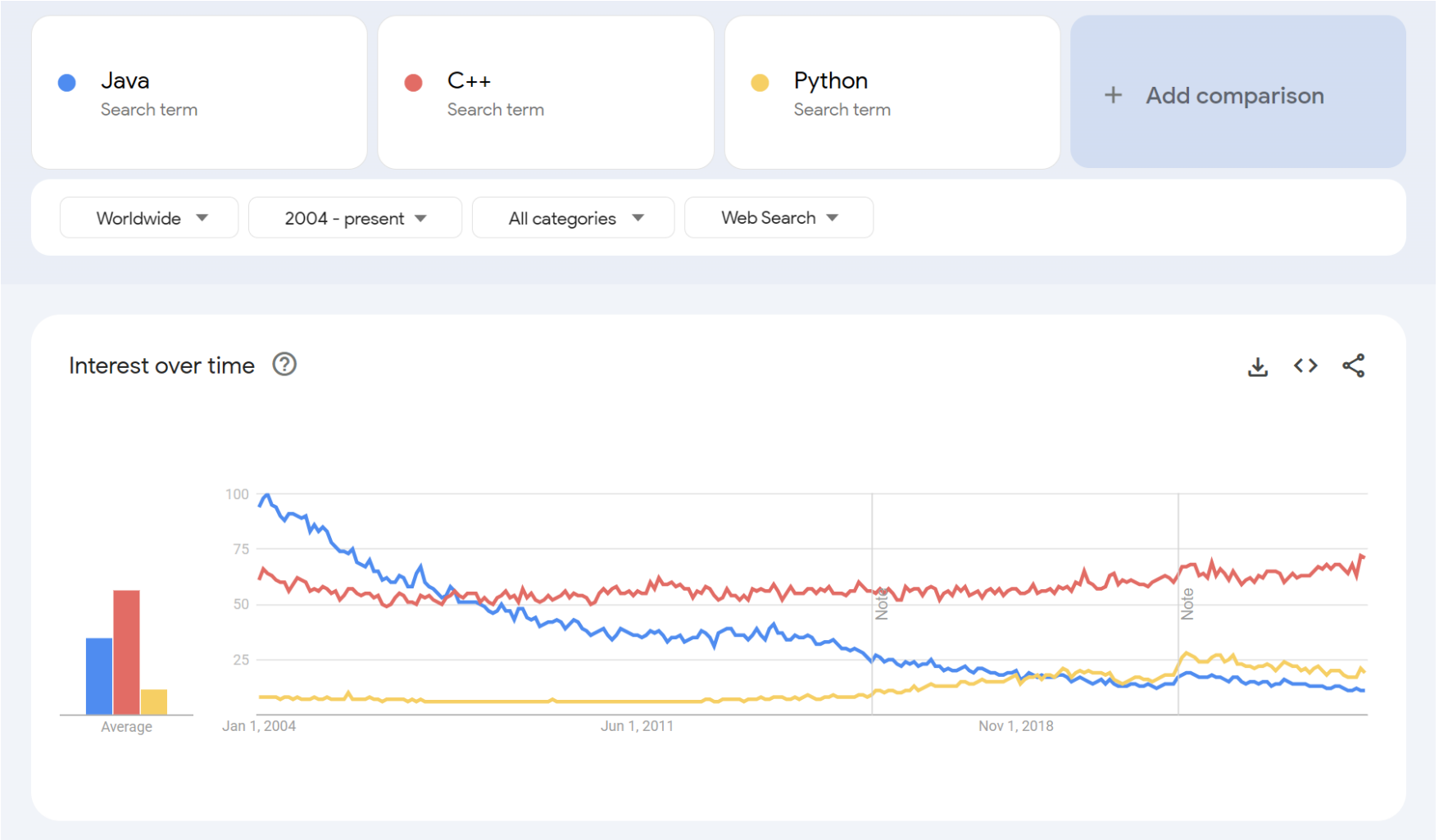
- Java ist ausführlich (verbose), gut lesbar und klar strukturiert.
- Andere Programmiersprachen sind sehr nahe an Java, z.B. C#, Kotlin, Scala
  - Leichter Umstieg auf: C++, Python, Ruby, JavaScript, TypeScript (andersrum: Hölle!)
- Sehr große Community und viel Support.
- Reichhaltige API und viele Open-Source-Bibliotheken.
- Zahlreiche Entwicklungswerkzeuge (IDEs, Debugging, Monitoring).
- Plattformunabhängig (läuft auf Windows, Mac, Linux, Android …).
- Die meisten aktiven Repositories und Entwickler auf GitHub (unter allen Sprachen).
- Hat Beständigkeit („Staying Power“).
- Reif und vielseitig.
- Anfängerfreundlich, lieb ☺.

Bekannte Java Beispiele





Was sagt Google Trends?



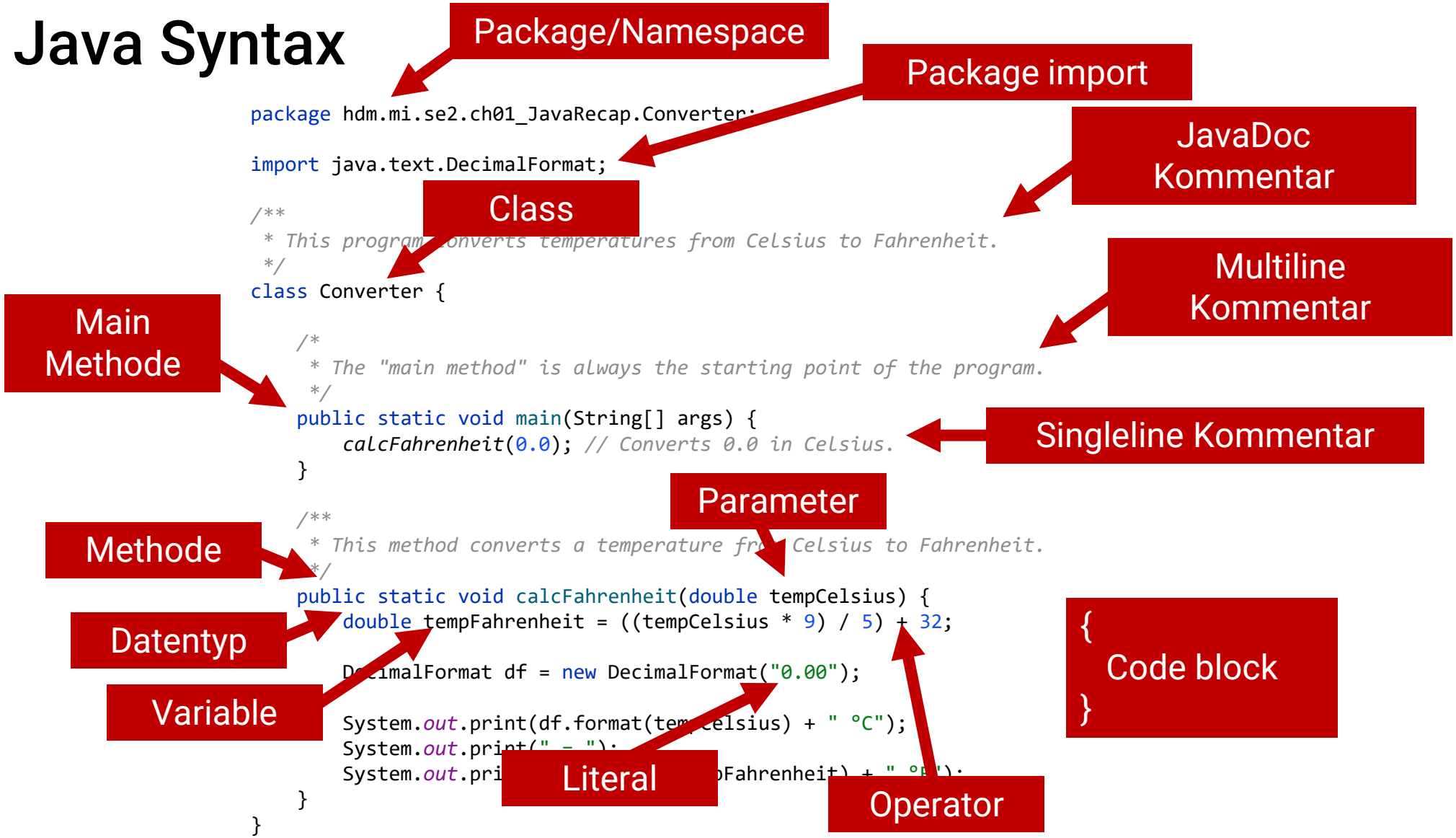
Kritik an Java

- Schlechte Unterstützung für moderne Features wie Streams und Lambdas (zumindest lange Zeit; erst ab Java 8 besser).
- Neuer 6-Monats-Release-Zyklus → viele Updates, manchmal schwer mitzuhalten.
- Sehr viel Boilerplate-Code (wiederholender Standardcode).
- Java (JVM) gilt als langsamer und speicherintensiver im Vergleich zu nativ kompilierten Sprachen wie z.B C++.
- GUIs wirken oft altmodisch und wenig attraktiv.
- Performance-Probleme, hoher Speicherverbrauch.
- Code kann komplex und „überladen“ wirken.
- „Alt, nicht cool“.

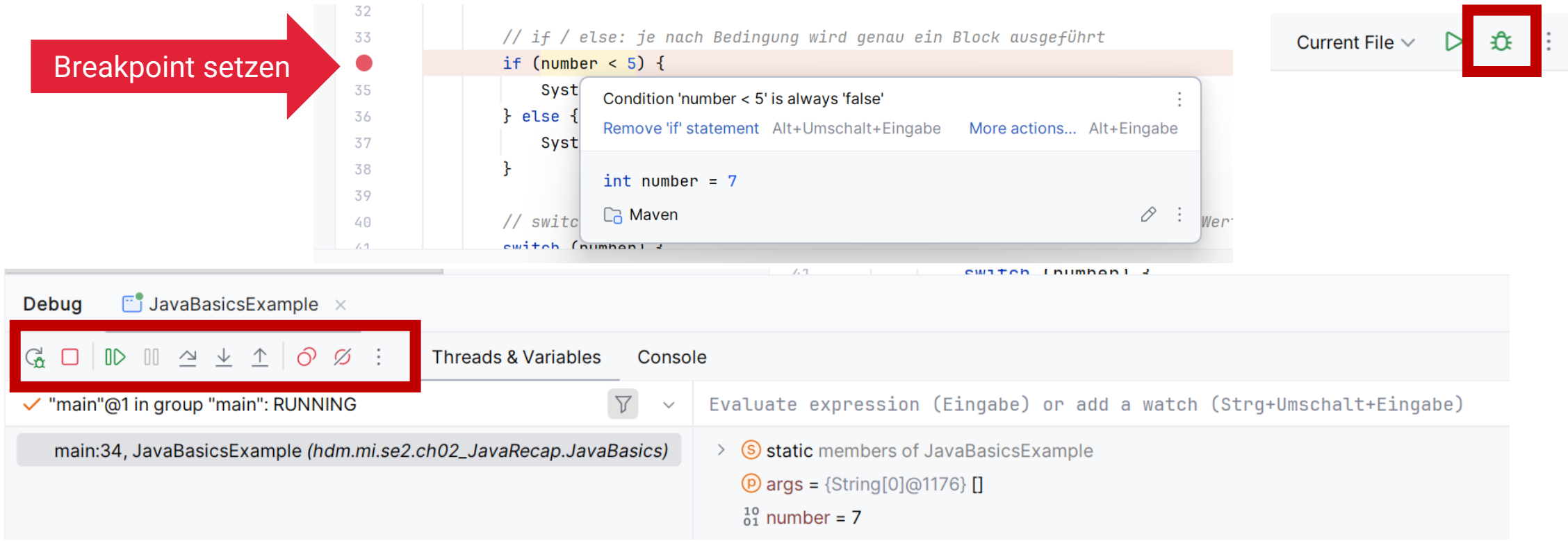
Was Java nicht kann

- Keine signed integer types (wie `unsigned int` in C++).
- Keine Operatorüberladung (außer + und += bei Strings).
- Keine structs (nur Klassen).
- Kein echtes funktionales Programmieren (z. B. keine echten Funktionen als Parameter in frühen Versionen) → Lernen wir in „Streams“ noch kennen!
- Arrays haben maximale Länge von $2^{31} - 1$ → kann zu Compilierungs-Fehlern führen.
  - Ungeeignet für viele Algorithmen (z.B. Machine Learning, Rendering, Finite Element …)
- Probleme bei Integration von Primitiven und Arrays.
- Sicherheitsprobleme bei Parallelisierung und Serialisierung.
- Generics verwenden Type Erasure → generische Typinformationen gehen zur Laufzeit verloren (z. B. `List<String>` und `List<Integer>` sehen zur Laufzeit gleich aus).

Java Syntax


Package/Namespace
Package import
JavaDoc Kommentar
Class
Multiline Kommentar
Main Methode
Singleline Kommentar
Parameter
Methode
Datentyp
Variable
{ Code block }
Literal
Operator
package hdm.mi.se2.ch01_JavaRecap.Converter;
import java.text.DecimalFormat;
/**
* This program converts temperatures from Celsius to Fahrenheit.
*/
class Converter {
/*
* The "main method" is always the starting point of the program.
*/
public static void main(String[] args) {
calcFahrenheit(0.0); // Converts 0.0 in Celsius.
}
/**
* This method converts a temperature from Celsius to Fahrenheit.
*/
public static void calcFahrenheit(double tempCelsius) {
double tempFahrenheit = ((tempCelsius * 9) / 5) + 32;
DecimalFormat df = new DecimalFormat("0.00");
System.out.print(df.format(tempCelsius) + " °C");
System.out.print(" = ");
System.out.print(df.format(tempFahrenheit) + " °F");
}
}

Java Debugging



- Programmausführung Zeile für Zeile nachvollziehen
- Werte, Zustände und Kontrollfluss überprüfen

Java Comments

- Dokumentiert **Schnittstellen und Verträge – ohne dass man den Code lesen muss**
- Ermöglicht **IDE-Tooltips und automatische API-Dokumentation**
- Erleichtert **Nutzung, Wartung und Weiterentwicklung** von Code

```
// Single line comment

/* Multi line
   comment */
/* Also a multi line
 * Comment */
```

```
/**
* This is a JavaDoc (or Doc) comment -> can be extracted to HTML using javadoc
*/
```

Verwenden Sie dies am besten gleich durchgehend für die Beschreibung Ihres Codes

JavaDoc Comments

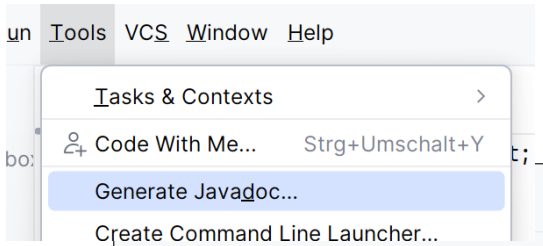
```java
/**
 * Repräsentiert ein einfaches Bankkonto mit grundlegenden Operationen.
 *
 * <p>Diese Klasse dient als Beispiel für die Dokumentation mit JavaDoc im Modul
 * Software-Entwicklung 2.</p>
 *
 * @author Valentin Schwind
 */
public class BankAccountExample {

    /** Eindeutige Kontonummer. */
    private final String accountNumber;

    /** Aktueller Kontostand. */
    private double balance;

    /**
     * Erzeugt ein neues Bankkonto mit einem Anfangsguthaben.
     *
     * @param accountNumber die eindeutige Kontonummer
     * @param initialBalance das Anfangsguthaben (muss &gt;= 0 sein)
     * @throws IllegalArgumentException wenn das Anfangsguthaben negativ ist
     */
    public BankAccountExample(String accountNumber, double initialBalance) {
        if (initialBalance < 0) {
            throw new IllegalArgumentException("Das Anfangsguthaben darf nicht negativ sein");
        }
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }
```

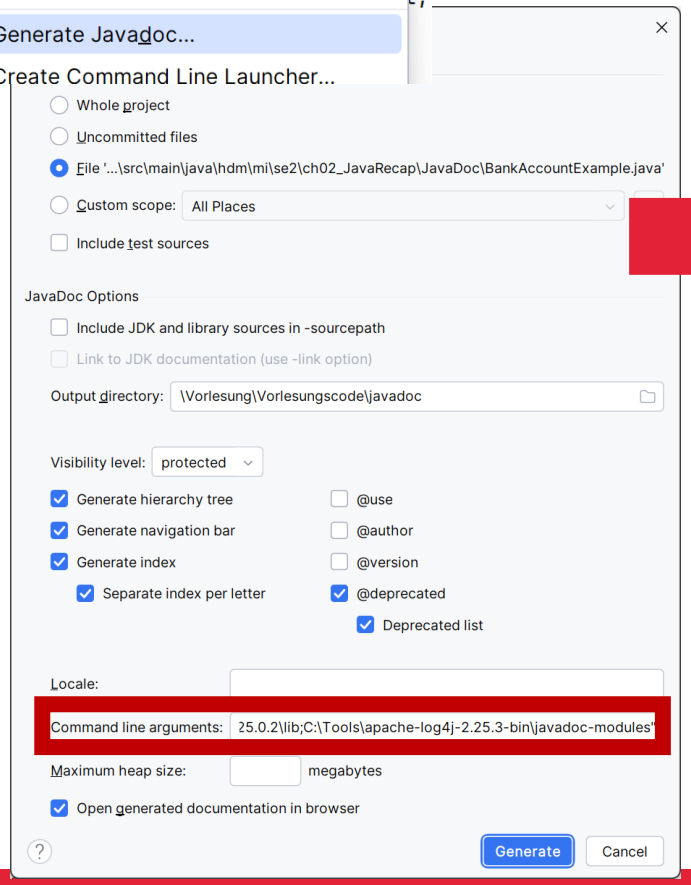
Klassenbeschreibung

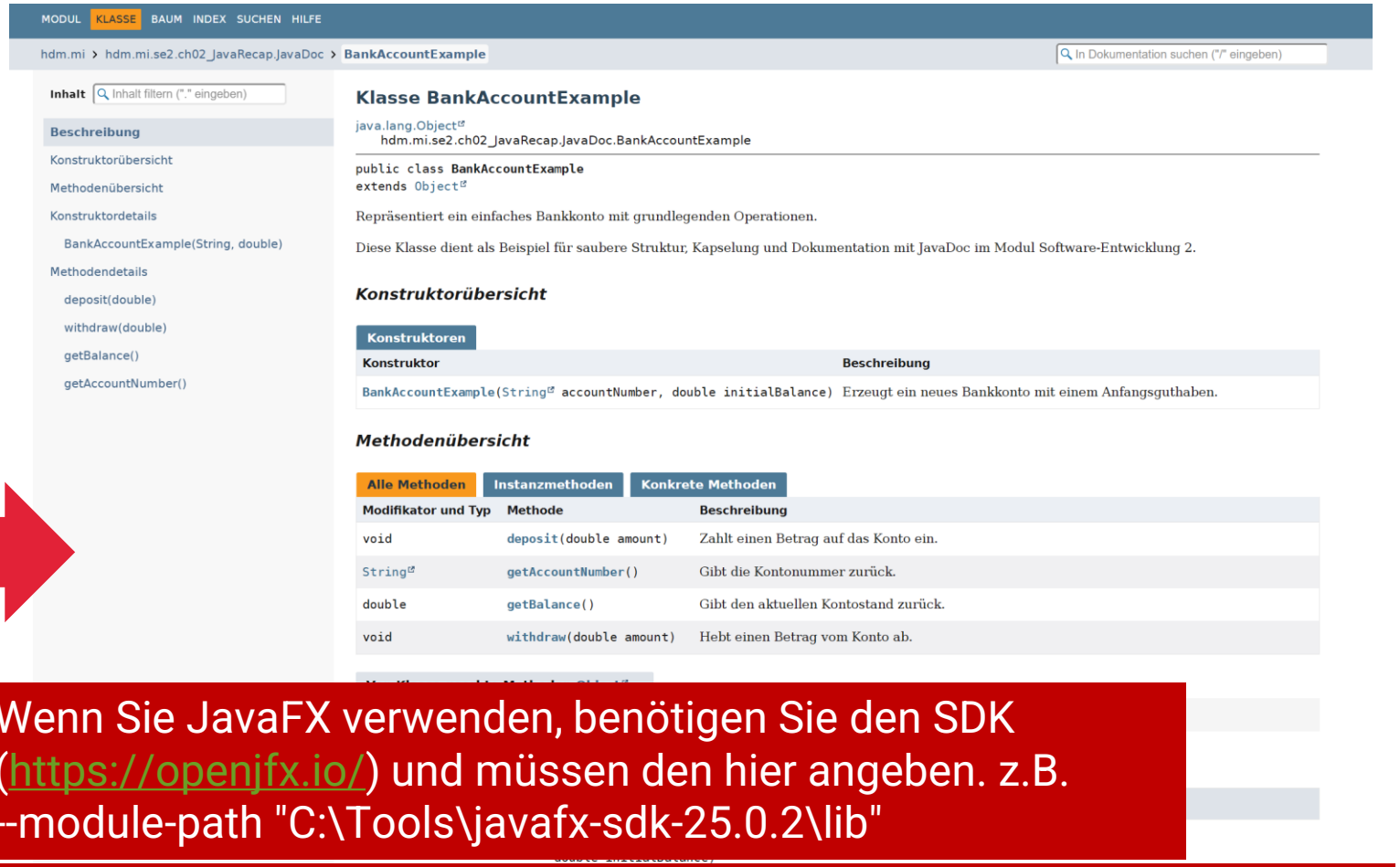
Feldbeschreibung

Methodenbeschreibung

JavaDoc







Wenn Sie JavaFX verwenden, benötigen Sie den SDK (https://openjfx.io/) und müssen den hier angeben. z.B.
--module-path "C:\Tools\javafx-sdk-25.0.2\lib"

Wenn Sie Log4j verwenden, benötigen Sie den SDK (https://logging.apache.org/log4j/2.x/download.html) und müssen den ebenfalls hier angeben (mit „;“ von JavaFX trennen). z.B.
--module-path "C:\Tools\apache-log4j-2.25.3-bin" (nur API und Core)

Über die Java Virtual Machine (JVM)

- Portabilität wird durch eine **Zwischendarstellung** erreicht: den **Java-Bytecode**.
  - Der Java-Compiler (javac) übersetzt Quellcode (.java) in Bytecode (.class). Die JVM führt diesen Bytecode plattformunabhängig aus.
- Bytecode ist **nicht** architekturspezifischer Maschinencode.
- Bytecode wird von einer **virtuellen Maschine (JVM)** ausgeführt.
- Endnutzer brauchen dazu die **Java Runtime Environment (JRE)**, die eine JVM und Bibliotheken enthält.
- Die JVM ist ein abstrakter, virtueller Computer, der nur durch eine **Spezifikation** definiert ist.

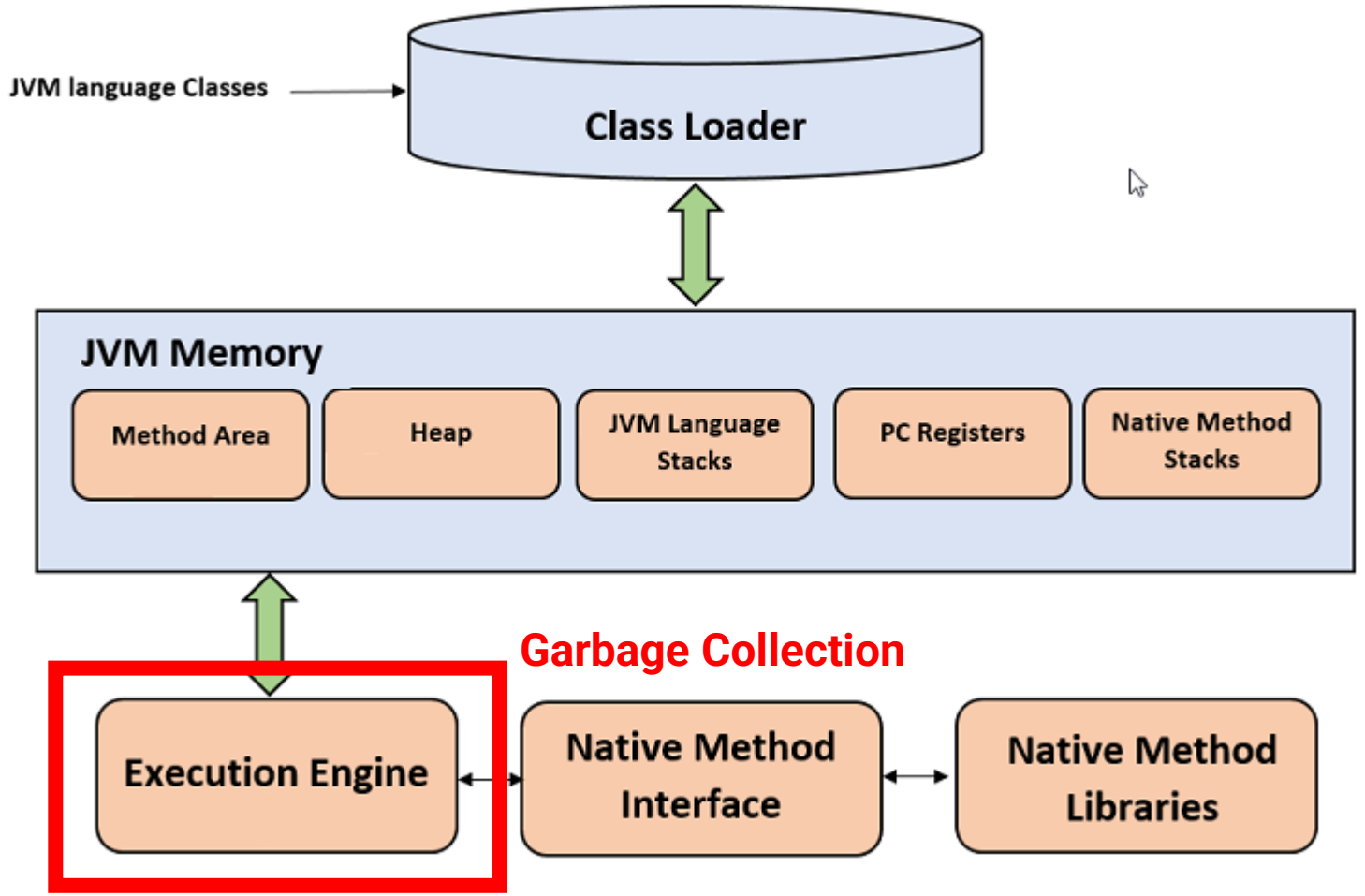
Java Virtual Machine (JVM)

- Die **JVM lädt und führt Bytecode aus** (`.class`-Dateien → oft gebündelt in `.jar`)
  - ist aber nur eine Spezifikation.
- Verschiedene Anbieter implementieren sie unterschiedlich:
  - HotSpot (Oracle), JamVM, Jikes, Dalvik/ART (Android)
- Bestandteile einer JVM:
  - **Class Loader** (inkl. Linker): Lädt Klassen und verbindet sie.
  - **Memory Model**: Verwalten von Speicher (Heap, Stack, etc.).
  - **Garbage Collector**: Automatische Speicherbereinigung.
  - **Adaptive Optimizer** (dynamische Neukompilierung, JIT).
- JVM **kann optimiert sein** für:
  - Client (schneller Start, kurze Laufzeiten)
  - Server (optimiert für lange Laufzeiten, hohe Performance).

Java Development Kit (JDK)

- **Java Development Kit (JDK)**
- Notwendig, wenn man Java-Programme **entwickeln** will.
- Enthält:
  - **JVM** (Java Virtual Machine)
  - **JRE** (Java Runtime Environment)
  - **Tools** zum Entwickeln, Debuggen und Überwachen von Java-Programmen.
- Unterschiede:
  - **JRE = JVM + Bibliotheken** (um Java-Programme **auszuführen**)
  - **JDK = JRE + Entwicklungswerkzeuge** (um Java-Programme **zu schreiben** und zu testen).

JVM Architektur



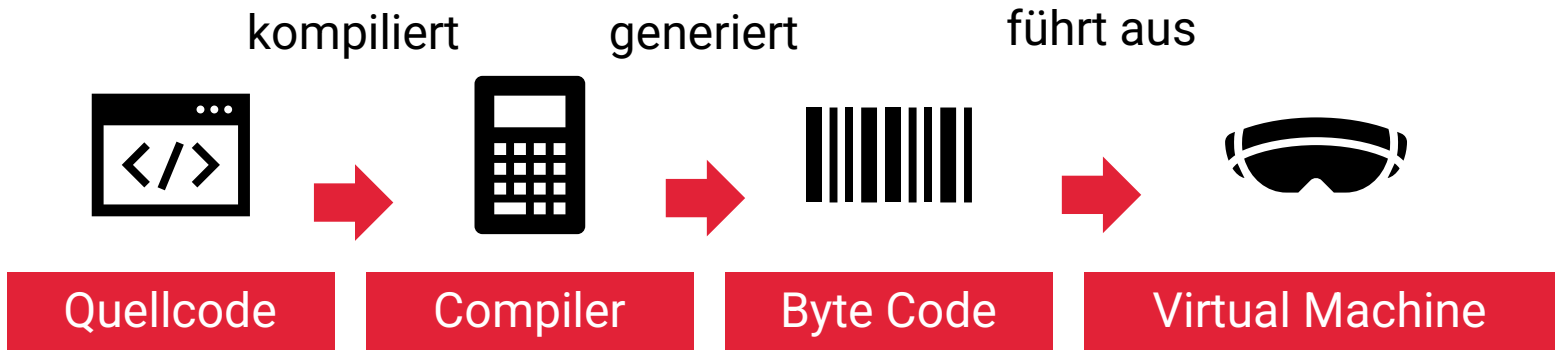
Source: https://howtodoinjava.com/java/basics/jdk-jre-jvm/

Garbage Collection

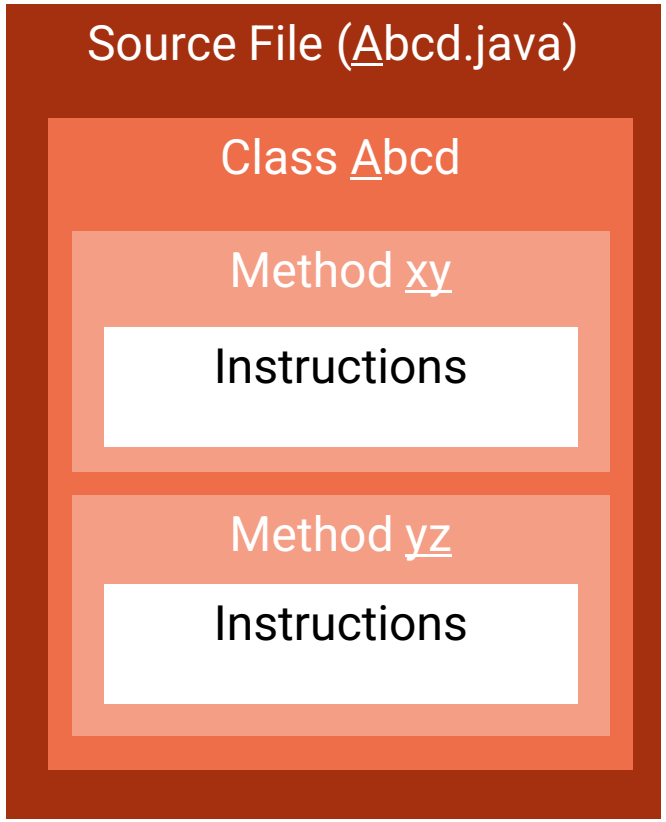
- Der Garbage Collector wird von der JVM Execution Engine gesteuert.
  - Java führt damit **automatisches Speichermanagement** durch.
- Wenn Programme laufen, werden Objekte im **Heap** erstellt.
- Speicherbereiche im Heap:
  - **Eden Space** → neue Objekte
  - **Survivor Space** → Objekte, die den ersten GC überstehen
  - **Tenured/Old Generation** → langlebige Objekte
  - **Permanent Generation (PermGen)** → Metadaten (in neueren Versionen ersetzt durch **Metaspace**)
- Wenn Objekte nicht mehr referenziert werden, gelten sie als „tot“ → der GC entfernt sie.
- Es gibt verschiedene Implementierungen von GC (z. B. Serial, Parallel, G1, ZGC).
- Der GC wird aktiv, wenn bestimmte Schwellenwerte überschritten werden (z. B. Speicher).

Java Code Ausführung

1. Code schreiben in einer IDE (z. B. Eclipse, IntelliJ).
2. Kompilieren in Bytecode (`.class`).
3. Ausführen auf jeder Plattform/ jedem Betriebssystem durch die JVM.
4. JVM übersetzt Bytecode in nativen Maschinencode.



Code Struktur in Java



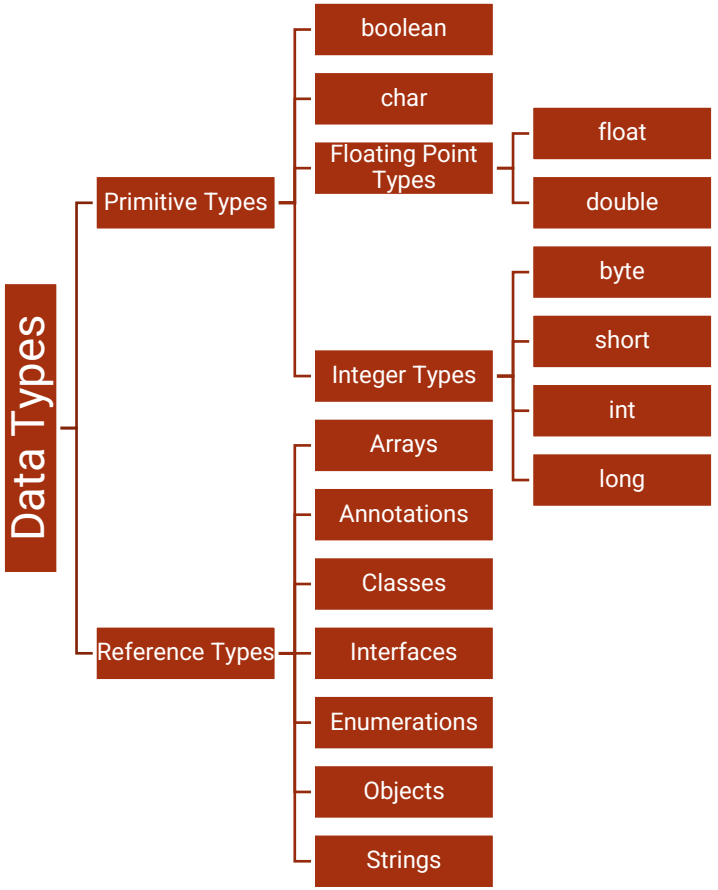
- Eine Source-Datei (`.java`) enthält die Definition einer oder mehrerer Klassen.
- Eine Klasse kann eine oder mehrere Methoden enthalten.
- Jede Methode besteht aus Anweisungen innerhalb eines Codeblocks `{ ... }`.
- Jeder Code muss in einer Klasse stehen (es gibt keine freien Funktionen wie in C).
- Eine Klasse kann nur mit einer `main()`-Methode gestartet werden.

Primitive Datentypen

- Java hat **acht primitive Datentypen**:
  - `byte, short, int, long` → Ganzzahlen
  - `float, double` → Kommazahlen
  - `boolean` → Wahr/Falsch
  - `char` → einzelnes Zeichen
- Primitive Werte werden **direkt im Speicher** abgelegt (nicht als Objekte referenziert).
- **String ist kein primitiver Datentyp**, sondern eine Referenz (`java.lang.String`).
- Zeichenketten in Anführungszeichen "..." erzeugen automatisch ein String-Objekt:
  - `String str1 = "bla";`
- Strings sind **immutable** (unveränderbar).
- Achtung: Unterschied zwischen primitiven Datentypen und Objekten („Wrapper“-Klassen).
  - z. B. `int` vs. `Integer`

Datentypen in Java

- Es gibt zwei Kategorien:
  - **Primitive Datentypen** (`int, float, double, boolean, …`)
  - **Nicht-primitive** / **Referenztypen** (Objekte, Arrays, Strings, eigene Klassen)
- Referenztypen werden über **Referenzen** (Adressen) gehandhabt.
- Arten von Referenzen:
  - **Strong Reference** (Standard): `Object obj = new Object();`
  - **Weak Reference** (`java.lang.ref.WeakReference`):
    - Wird z. B. in WeakHashMap genutzt.
    - Objekt kann vom GC gelöscht werden, auch wenn noch eine Referenz existiert.
  - **Soft Reference** (`java.lang.ref.SoftReference`):
    - Objekt wird gelöscht, wenn JVM Speicherprobleme bekommt.
  - **Phantom Reference** (`java.lang.ref.PhantomReference`):
    - Objekt wird nach `finalize()` in eine Referenz-Queue gelegt.



Definitionen

Data Type	Size	Value range	Detail	Default	Wrapper
boolean	1 bit	true / false	True or false	False	java.lang.Boolean
char	16 bit	0 ... 65.535 (z. B. 'A')	two complement integer	0	ava.lang.Character
byte	8 bit	-128 ... 127			va.lang.Byte
short	16 bit			0	java.lang.Short
int	32 bit	... 2.147.483.647	two complement integer	0	java.lang.Integer
long	64 bit	-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807	two complement integer	0	java.lang.Long
float	32 bit	+/-1,4E-45 ... +/-3,4E+38	IEEE 754 floating point	0.0	java.lang.Float
double	64 bit	+/-4,9E-324 ... +/-1,7E+308	IEEE 754 floating point	0.0	java.lang.Double

Gewährleistet auf allen Plattformen.
Keine zufälligen Änderungen.

Typsicherheit

- Java ist so entworfen, dass es **Typsicherheit erzwingt**.
  - Alles in Java passiert innerhalb von Objekten, und jedes Objekt ist eine Instanz einer Klasse.
  - Bevor ein Objekt verwendet werden kann, muss es korrekt angelegt (allocated) sein.
  - **Java erlaubt primitive Typen**, aber nur innerhalb gültiger Objekte.
  - Sogar **Zahlen haben explizite Typen**

    → z. B. double (mit d/D), float (mit f/F), long (mit l/L).
- Beispiel (Ariane 5, 1996):
  - Beim Start der Rakete führte eine fehlerhafte Typkonvertierung (64-bit Float → 16-bit Integer) zu einem Operand-Error.
  - Rakete explodierte nach 37 Sekunden (Schaden von 370 Mio. USD.)

    → zeigt, wie gefährlich fehlende Typsicherheit sein kann.

Typsicherheit

JavaScript (nicht typsicher)

```
var number = 10; // numeric value
var string = "10"; // string value
var sum = number + string; // numeric + string
```

```
1010
```

Was ist var???

Die Ausgabe ist die Verkettung von Zahl und Zeichenfolge.

Java (typsicher)

```
int number = 10; // numeric value
String string = "10"; // string value
int sum = number + string; // numeric + string
```

```
Error: Type mismatch: cannot convert from
String to int
```

Der Compiler hat die Typen überprüft und eine Ausnahme zur Kompilierzeit ausgelöst

Der Typ `var` in Java

```java
public class JavaVarExample {
    public static void main(String[] args) {

        // Verwendung von 'var' (seit Java 10)
        var result = 10 + 20; // Compiler leitet 'int' ab
        System.out.println(result);

        // Problem 1: Unklarer Typ und unklare Eigenschaften (z. B. Mutabilität)
        var data = getData();

        // Kompiliert, führt aber zur Laufzeit zu einer Exception
        data.add("four"); // UnsupportedOperationException

        // Problem 2: Veränderung des Typs
        var value = 1; // int
        value = 1.0; // Compilerfehler, kompiliert nicht
    }

    static List<String> getData() {
        // Liefert eine unveränderliche Liste, add, remove, clear sind verboten!
        return List.of("one", "two", "three");
    }
}
```

Der Compiler hat die Typen überprüft, fand den gut, hat aber eine Exception ausgelöst (der Code läuft weiter)

Hier ist passt der Wert nicht mehr zum Datentyp und der Code kompiliert nicht mehr.

`var` oder nicht `var`?

- `var` existiert, um lokalen Code kompakter zu machen. Java wird nicht von seiner statischen Typisierung gelöst.
  - `var` ist eine **lokale Typinferenz** (seit Java 10).
  - Die Typinformation wird **versteckt**.
  - Der Typ wird zur Compile-Zeit **immer noch eindeutig** aus dem Initialisierer abgeleitet.
- **ABER** der Typ ist im Quelltext nicht mehr sichtbar!
  - Teamarbeit wird erschwert.
  - Leser müssen den Datentyp rekonstruieren.
  - **Verwenden Sie in Ihren Projekten kein `var`.**

Warum Typsicherheit?

- **Vermeidung ungültiger Operationen**
  - Beispiel: `3 / "Hello, World"` ist nicht erlaubt (arithmetische Regeln definieren sowas nicht).
- **Speichersicherheit**
  - Keine wilden Pointer wie in C → kein Zugriff auf beliebige Speicheradressen.
  - Schutz vor Buffer Overflows durch automatisches Speicher-Management.
- **Fehlerprävention** bei Logik & Semantik
  - z. B. Typkonvertierungsfehler vermeiden.
  - Einheitenfehler verhindern (z. B. imperial → metrisch).

Mars Climate Orbiter (MCO)

Typsicherheit und Casting

- `true` und `false` sind nicht gleich `1` und `0`.
- Es gibt **kein automatisches Casting** zwischen inkompatiblen Typen.
- Beispiel:

```
int x = 1;
while(x) { };
```

  …ist nicht erlaubt, weil `int ≠ boolean`.

  - Auch `(Boolean) x` funktioniert nicht.
- **Numerische Typen** können automatisch umgewandelt werden (z. B. int → double), aber nur in einer bestimmten Kette.
- Umgekehrt (z. B. `double` → `int`) ist nur mit **explizitem Cast** möglich.

Type Casting

- **Widening Casting (automatisch):** Kleinere Typen → größere Typen
  - byte → short → int → long → float → double
    char ↗

```
int myInt = 9;
double myDouble = myInt; // Automatic casting: int to double
```

- **Narrowing Casting (manuell):** Größere Typen → kleinere Typen
  - double → float → long → int → char → short → byte

```
double myDouble = 9.78;
int myInt = (int) myDouble; // Manual casting: double to int
```

Autoboxing und Unboxing

- Java hat ein zweistufiges Typsystem:
  - **Primitive Types**: int, boolean, double, …
  - **Reference Types**: Integer, Boolean, Double, … (können auch `null` sein)
- Jeder primitive Typ hat eine **Wrapper-Klasse**.
- Java kann zwischen primitiven Typen und ihren Wrappern konvertieren:
  - Dieser Vorgang heißt **Autoboxing** (primitive → Wrapper)
  - Umgekehrt: **Unboxing** (Wrapper → primitive)

→ Literal `null`/`NULL`: ein Wert, der eine leere Referenz angibt

```
int a = 1;          // primitive type
Integer b = a;      // reference type: boxing
int c = b;          // primitive type: unboxing
int e = a + b + c; // 3
```

Entsprechende Primitive-/Referenztypen

Primitiver Typ	Initialisierung	Referenztyp	Initialisierung
bool	`bool b = false;`	Bool	`Bool b = false;`
byte	`byte b = 1;`	Byte	`Byte b = 1;`
char	`char c = 'a';`	Character	`Character c = 'a'`
short	`byte s = 1;`	Short	`Short s = 1;`
int	`int i = 1;`	Integer	`Integer i = 1;`
long	`long l = 0;`	Long	`Long l = 0l;`
float	`float f = 0;` `float f = 0.1f;`	Float	`Float f = 0f;` `Float f = 0.1f;`
double	`double d = 0;` `double d = 0.1d;`	Double	`Double d = 0;` `Double d = 0.1d;`

Groß- und Kleinbuchstaben sind erlaubt!

Mehrdeutige Methodenaufrufe

```java
public static void main(String[] args) {
    int test = 20;
    myMethod(test); // I am primitive long
}
static void myMethod(long i) { System.out.println("I am primitive long"); }
static void myMethod(Integer i) { System.out.println("I am an Integer Object"); }
```

- Es ist nicht ratsam, Autoboxing und Unboxing für seriöse Rechnungen oder anderen leistungsabhängigen numerischen Code zu verwenden.
  - Warum?
- **Ein `Integer` ist kein Ersatz für ein `int`**: Autoboxing und Unboxing verwischen nur die Unterscheidung zwischen primitiven Typen und Referenztypen, die benötigt werden
  - z.B. in Collections (unterstützen keine Primitive)

Was bedeutet das?

```
Long sum = 0l; // reference type
for (long k = 0; k < Integer.MAX_VALUE; k++) {
    sum += k;  // highly inefficient!!!
}
```

- Problem: Bei jedem Durchlauf findet **Autoboxing** statt:
  - `sum` ist ein `Long`-Objekt, keine primitiver Datentyp `long`.
  - Der Ausdruck `sum += k;` bedeutet:
    - `Long` → `long` (Unboxing)
    - Addition durchführen
    - Ergebnis zurück zu `Long` (Boxing)
  - Das passiert Milliarden Mal → katastrophale Performance!

Strings

- Ein `String` ist eine Folge von Zeichen (`char`)
- Der einzige Objektdatentyp in Java mit einem überladenen Operator: “+“
- Kann mit oder ohne `new` erzeugt werden

```
String str1 = new String("Hello");
String str2 = new String("World");
System.out.println(str1 + " " + str2); // Hello World

String str1 = "Hello";
String str2 = "World";
System.out.println(str1 + " " + str2); // Hello World
```

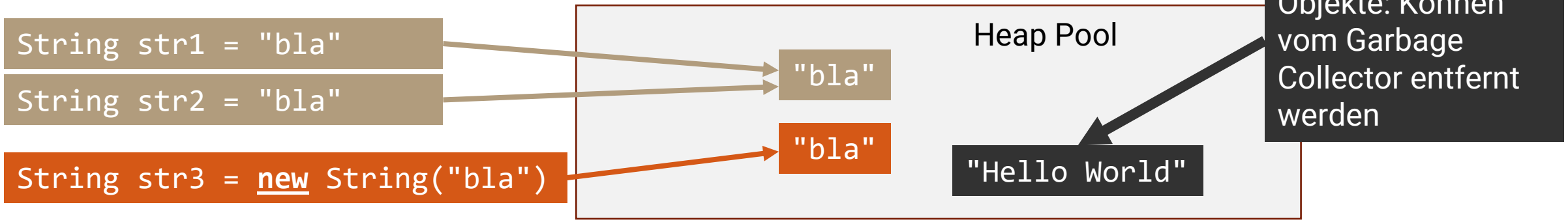
- Unveränderlicher („immutable“) Referenzdatentyp, einmal erstellt, kann der Inhalt nicht verändert werden.
  - Hä? Aber ich kann doch Strings ändern?

Java Heap Pool

```java
String str1 = "bla";
String str2 = "bla";

if(str1 == str2) System.out.println("true");
else System.out.println("false");
// true
```

- Warum ist die Ausgabe `true`? Wir haben doch verschiedene Objekte!?

→ Weil zwei verschiedene Variablen auf **dasselbe Objekt aus dem Java-Heap-Pool** verweisen und sich somit dieselbe Speicherressource teilen.



Vergleichen von Referenzen

```
String str1 = new String("bla");
String str2 = new String("bla");

if(str1 == str2) System.out.println("true");
else System.out.println("false");
// false
```

- **Verwenden Sie die Methode `equals()`!**
  - Sie vergleicht den Inhalt des Objekts

```
if(str1.equals(str2)) System.out.println("true");
else System.out.println("false");
// true
```

- **Wichtig**: In Java können Sie keine Operatoren wie == überschreiben (in C++ geht das!)
  - Nur Methoden wie `equals()`.
  - Wir schreiben in Java oft unsere eigenen `equals`-Methoden, um Objektvergleiche zu machen!

Heap Pool & Speicherverwaltung

- Strings, die als **Literal** (`"bla"`) erstellt werden, landen im **String Pool**.
- Wird `new String("bla")` verwendet, wird **ein neues Objekt im Heap** angelegt.
- **Heap Memory** besteht aus:
  - **Eden Space**: neuer Speicher für Objekte.
  - **Survivor Space**: Objekte, die GC überleben.
  - **Tenured Generation / Old Gen**: langlebige Objekte.
- **Non-Heap**: für Konstanten und Metadaten (PermGen / Metaspace).
  - **Permanent**: Konstanten und reflektierende Daten der virtuellen Maschine
  - **Coda Cache**: HotSpot enthält einen Code-Cache, der Speicher enthält, der für die Neukompilierung und Speicherung von systemeigenem Code verwendet wird.

Äquivalenzrelationen

- `==` ist ein Referenzvergleich (wenn beide Objekte auf den gleichen Speicherplatz zeigen)
- `equals()` wertet in der Regel den Vergleich von Werten in den Objekten aus (wenn beide Objekte den gleichen Inhalt haben)
- Die Verwendung von `equals()` basiert auf der Konvention: Für die Objekte sollten die Kopien `o1`, `o2`, `o3`, `equals()` die folgenden **Äquivalenzrelationen** erfüllen:
  - **Reflexivität**: `o1.equals(o1) == true`
  - **Symmetrie**: `o1.equals(o2) == o2.equals(o1)`
  - **Transitivität**: Wenn `o1.equals(o2)` und `o2.equals(o3)`, dann auch `o1.equals(o3)`

StringBuilder and StringBuffer

- Echt jetzt? Strings greifen immer zufälliges Zeug aus dem HeapPool!?!?
  - Keine Panik: Es gibt **StringBuilder** oder **StringBuffer**!
- **StringBuilder**
  - String-Objekte, die geändert werden können (nicht mehr „unveränderlich“)
    - `StringBuilder sb = new StringBuilder();`
  - Zusätzliche Funktionen zum Bearbeiten von Zeichenfolgen (Anhängen, Einfügen, Umkehren,...)
  - Ordnet mehr Speicher zu als benötigt (`capacity()`) mit zusätzlichen 16 leeren Elementen
  - Nicht threadsicher
- **StringBuffer**
  - Fast identisch mit StringBuilder: `StringBuffer sb = new StringBuffer();`
  - Weniger Leistung
  - Threadsafe

Das Keyword `final`

- `final`-Variablen sind Konstanten (`const` in C++), feste Werte
- `final`-Variablen können nur **einmal initialisiert** werden, müssen aber zum Zeitpunkt der Deklaration nicht initialisiert sein ("blank final")

  → kann z.B. innerhalb eines Konstruktors initialisiert werden

  → Es gibt jedoch keine direkte Initialisierung wie in C++
- Bitte beachten Sie:
  - Eine `final`-Methode kann nicht überschrieben werden
  - Eine `final`-Methode kann nicht durch Unterklassen ausgeblendet werden
  - Eine `final`-Klasse kann nicht untergeordnet werden

Java Operatoren

- Zum Ausführen von Operationen für Variablen und Werte
- Folgende Gruppen:
  - Arithmetische Operatoren (`+`, `-`, `*`, `/`, `%`, `++`, `--`)
  - Zuweisungs-Operatoren (`=`, `+=`, `-=`, …)
  - Vergleichsoperatoren (`==`, `!=`, `>`, `<`, `<=`, `>=`)
  - Logische Operatoren (`&&`, `||`, `!`)
  - Bitweise Operatoren (`&`, `|`, `~`, `^`, `<<`, `>>`, `>>>`)

Kontrollstrukturen

Bedingungen

if/else

```
if (a > 0) {
    a = a + 1;
} else {
    a = a - 1;
}
```

switch

```
switch (a) {
    case 0: a = a + 1; break;
    case 1: a = a + 1; break;
    default: a = a - 1;
}
```

Schleifen

while

```
while (a > 0) {
    a--;
}
```

do … while

```
do {
    a--;
} while (a > 0);
```

for

```
for (int i = 0; i < a; i++) {
    b *= 2;
}
```

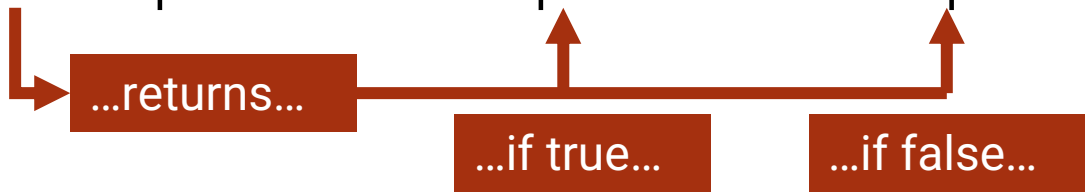
Bedingungen

- `if` verwendet logische Bedingungen (`a < b, a == b,...`) aus der Mathematik
- `if` wird verwendet, um einen Codeblock anzugeben, der ausgeführt werden soll, wenn eine angegebene Bedingung wahr ist
- `else`, um einen Codeblock anzugeben, der ausgeführt werden soll, wenn dieselbe Bedingung falsch ist
- `else if`, um eine neue zu testende Bedingung anzugeben, wenn die erste Bedingung falsch ist
- `switch`, um viele alternative Codeblöcke anzugeben

Der Ternär-Operator

- **Der ternäre Operator besteht aus**
  - **einer Bedingung**, die entweder als wahr oder falsch ausgewertet wird (`booleanExpression`)
  - **einem Wert**, der zurückgegeben wird, wenn die Bedingung wahr ist (`expression1`), und ein weiterer Wert, der zurückgegeben wird, wenn die Bedingung falsch ist (`expression2`)

```
booleanExpression ? expression1 : expression2
```



Beispiel: Maximum zurückgeben

```
int val1 = 200; int val2 = 100;
int max = val1 >= val2 ? val1 : val2;
System.out.println(max); // 200
```

Beispiel: Rückgabe von NULL, wenn das Objekt NULL ist

```
String value = bla != null ? bla.toString() : null;
System.out.println(value); // null
```

~~Funktionen~~ Methoden in Java

- Java ist eine **rein objektorientierte Sprache** → es gibt **keine freien Funktionen**.
- Alles ist eine **Methode innerhalb einer Klasse**.

	ohne Parameter	mit Parameter

ohne Rückgabewert	`void doIt() {`
	`  System.out.print("bla");`
	`}`
		`void doIt(String _str) {`
		`  System.out.print(_str);`
		`}`

mit Rückgabewert	`String doIt() {`
	`  String str = "bla";`
	`  return str;`
	`}`
		`String doIt(String _str) {`
		`  String str = _str + "bla";`
		`  return str;`
		`}`

- **Parameter** definieren auch die **Signatur** einer Methode.
  - Unterschiedliche Signaturen ermöglichen **Methodenüberladung**

Überladen der Methode

- **Java weiß, welche Methode aufgerufen werden soll, indem es die Methodensignaturen überprüft**
- Das Verhalten von Java wird als **Kompilierzeitpolymorphismus** oder statische oder frühe Bindung („early binding“) bezeichnet

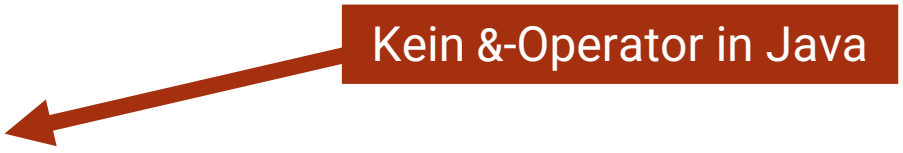
```
void doIt() {
    System.out.print("bla");
}

void doIt(String _str) {
    System.out.print(_str);
}

doIt(); // bla
doIt("Hello World"); // Hello World
```

Übergabemodelle

- Java hat ein einheitliches **Übergabemodell**:
  - Es wird immer nur eine **Kopie des Wertes** übergeben. **Das nennt man Call-by-Value**.
  - Änderungen innerhalb einer Methode wirken sich nicht auf die ursprünglichen Werte außerhalb aus.
  - Gilt sowohl für primitive Datentypen als auch für Referenztypen (Objekte).
- **Bei Objekten** wird eine **Kopie der Referenz** übergeben → dadurch kann man den Inhalt des Objekts ändern, aber nicht die Referenz selbst.
- **Objekte in Java werden mit Call-by-Value aufgerufen, wobei der Wert eine Referenz ist** (dies wird als "Aliasing" bezeichnet)
  - Immer noch Call-by-Value

Java: Immer Call-by-Value

```java
public class CallByValueExample {

    public static void main(String[] args) {
        String sb = new String("My first string");
        answer(sb);
        System.out.println(sb);
    }

    private static void answer(String sb) {
        sb = new String("My second string");
    }
}
```




```
My first string
```

- Java ist strikt Pass by Value!
- Aber was ist mit Objekten, die mithilfe von Referenzen behandelt werden?

C++: Call-by-Reference (Primitives)

Call-by-Value C++:

```cpp
#include <iostream>

void myMethod(int _i){ _i++; }

int main() {
    int i = 0;
    myMethod(i);
    std::cout << i; // Output: 0
}
```

Call-by-Value Java:

```java
public class CallByValueExample {
    private static void myMethod(int _i) { _i++; }

    public static void main(String[] args) {
        int i = 0;
        myMethod(i);
        System.out.println(i); // Output: 0
    }
}
```

Call-by-Reference C++:

```cpp
#include <iostream>

void myMethod(int&_i){ _i++; }

int main() {
    int i = 0;
    myMethod(i);
    std::cout << i; // Output: 1
}
```

&-Operator in C++

In Java nicht möglich

C++: Call-by-Reference (Objects)

Beispiel:

```cpp
#include <iostream>
#include <string>

using namespace std;

class Dog {
    private:
        string a;
    public:
        Dog(string _a){ a = _a;}
        string getName(){return a;}
};

static void foo(Dog& d) { // working on a reference
    cout << d.getName() << "\n"; // "Max"
    d = Dog("Fifi");
    cout << d.getName() << "\n"; // "Fifi"
}
```

```cpp
int main() {
    Dog newDog = Dog("Max");
    Dog oldDog = newDog;
    cout << newDog.getName() << "\n"; // "Max"
    cout << oldDog.getName() << "\n"; // "Max"
    cout << (&newDog == &oldDog) << "\n"; // 0 -> false

    cout << "\nDoing stuff in method with objects..." << "\n";
    foo(newDog);
    cout << "\nReturing into main..." << "\n";

    cout << newDog.getName() << "\n"; // "Fifi"
    cout << oldDog.getName() << "\n"; // "Max"
    cout << (&newDog == &oldDog); // 0 -> false
}
```

Manipulation von Objekten

Änderung!

In C++ können wir direkt an der Referenz arbeiten

Java: Call-by-Value (Objects)

Beispiel:

```
class Dog {
    private String a;
    public Dog(String a){ this.a = a;}
    String getName(){return a;}
}

public class CallByValu

    public static void foo(Dog d)  now on a copy
        System.out.println(d.getName()); // "Max"
        d = new Dog("Fifi");
        System.out.println(d.getName()); // "Fifi"
    }
```

```
public static void main(String[] args) {
    Dog newDog = new Dog("Max");
    Dog oldDog = newDog;
    System.out.println(newDog.getName()); // "Max"
    System.out.println(oldDog.getName()); // "Max"
    System.out.println(newDog == oldDog); // true

    System.out.println("\nDoing stuff in method with objects...");
    foo(newDog);
    System.out.println("\nReturing into main...");

    System.out.println(newDog.getName()); // "Max"
    System.out.println(oldDog.getName()); // "Max"
    System.out.println(newDog == oldDog); // true
}
}
```

Keine Objektmanipulation

& nicht möglich

Keine Änderung

In Java wird eine Referenz übergeben und **eine Kopie der Referenz** in foo() erstellt.
Außerhalb der Methode finden keine Änderungen statt, wir können das wahre Objekt nicht ändern!

Java: Und Object-Arrays?

Beispiel:

```
class Dog {
    private String a;
    public Dog(String a){ this.a = a;}
    String getName(){return a;}
}

public class CallByValueDogsExample {

    static void foo(Dog d) { // working on a copy
        System.out.println(d.getName()); // "Max"
        d = new Dog("Fifi");
        System.out.println(d.getName()); // "Fifi"
    }
```

```
public static void main(String[] args) {
    Dog[] newDog = new Dog[]{new Dog("Max")};
    Dog[] oldDog = newDog;
    System.out.println(newDog[0].getName()); // "Max"
    System.out.println(oldDog[0].getName()); // "Max"
    System.out.println(newDog == oldDog); // true

    System.out.println("\nDoing stuff in method with objects...");
    foo(newDog[0]);
    System.out.println("\nReturing into main...");

    System.out.println(newDog[0].getName()); // "Max"
    System.out.println(oldDog[0].getName()); // "Max"
    System.out.println(newDog == oldDog); // true
}
```

Einzelne Manipulation

Einzelne Variable

Auch hier wird eine Referenz kopiert, es ändert sich nichts.

Was passiert, wenn ich mit den einzelnen Verweisen im Array arbeite?

Java: Und Object-Arrays?

Beispiel:

Kopieren und bearbeiten wir das gesamte Array!

Array

Array-Manipulation

Array

```
class Dog {
    private String a;
    public Dog(String a){ this.a = a;}
    String getName(){return a;}
}

public cl                alueDogsExample {

    public stati      d foo(Dog[] d) {  rking on a copy
        System.out.println(d[0]      Name()); // "Max"
        d[0] = new Dog("Fi      );
        System.out.println(d[0].getName()); // "Fifi"
    }
}
```

```
ublic static void main(String[] args) {
    Dog[] newDog = new Dog[]{new Dog("Max")};
    Dog[] oldDog = newDog;
    System.out.println(oldDog[0].getName()); // "Max"
    System.out.println(newDog == oldDog); // true

    System.out.println("\nDoing stuff in method with objects...");
    foo(newDog);
    System.out.println("\nReturing into main...");

    System.out.println(newDog[0].getName()); // "Fifi"
    System.out.println(oldDog[0].getName()); // "Fifi"
    System.out.println(newDog == oldDog); // true
}
```

... wir ändern alle Arrays, die auf dieselbe Referenz zeigen!
(Dog Max ist verschwunden → Garbage Collector)

Java: Und Object-Arrays?

Example:

```
public static void main(String[] args) {
    Dog[] newDog = new Dog[]{new Dog("Max")};
    ldDog = newDog;
    out.println(newDog[0].getName()); // "Max"
    out.println(oldDog[0].getName()); // "Max"
    out.println(newDog == oldDog); // true

    System.out.println("\nDoing stuff in method with objects...");
    foo(newDog);
```

Aus der OOP-Perspektive machen wir hier völligen Unsinn.

Die Art und Weise, wie man in Java den Inhalt eines Objekts richtig manipuliert, besteht darin, **die Referenz des neuen Objekts zurückzugeben** (`return new`) oder mit dem Inhalt des Objekts zu arbeiten!

(Es gibt auch andere Lösungen. Aber wir sollten jetzt aufhören, Unsinn zu machen.)

Verwenden wir mal das `return`-Statement!

Example

Reguläres Objekt

Überschreiben der Referenz

Zurückgeben der neuen Referenz

```
class Dog {
    private String a;
    public Dog(String a){ this.a = a;}
    String getName(){return a;}
}

public class CallDogs {

    public static Dog foo(Dog d) {
        System.out.println(d.getName()); // "Max"
        d = new Dog("Fifi");
        System.out.println(d.getName()); //
        return d;
    }
}
```

```
public static void main(String[] args) {
    Dog newDog = new Dog("Max");
    Dog oldDog = newDog;
    System.out.println(newDog.getName()); // "Max"
    System.out.println(oldDog.getName()); // "Max"
    System.out.println(newDog == oldDog); // true

    System.out.println("\nDoing stuff in method with objects...");
    newDog = foo(newDog);
    System.out.println("\nReturing into main...");

    System.out.println(newDog.getName()); // "Fifi"
    System.out.println(oldDog.getName()); // "Max"
    System.out.println(newDog == oldDog); // false
}
}
```

Das ist es, was wir wollen, aber eigentlich können wir auch hier einfach eine Setter-Methode für "name" implementieren

Objektinhalte ändern mit Setter-Methoden

```java
class Dog {
    private String a;
    public Dog(String a){ this.a = a;}
    String getName(){return a;}
    public void setName(String name) { this.a = name; }
}

public class CallDogs {
    public static void main(String[] args) {
        Dog newDog = new Dog("Max");
        Dog oldDog = newDog;
        System.out.println(newDog.getName()); // "Max"
        System.out.println(oldDog.getName()); // "Max"
        System.out.println(newDog == oldDog); // true

        System.out.println("\nDoing stuff in method with objects...");
        newDog.setName("Fifi");
        System.out.println("\nReturing into main...");

        System.out.println(newDog.getName()); // "Fifi"
        System.out.println(oldDog.getName()); // "Max"
        System.out.println(newDog == oldDog); // true
    }
}
```

Name ändern

Machen wir eine schöne Set-Methode

Was ist mit dem Swapping von Objekten?

Bedenken Sie die OOP!

Java: Kein Swapping mit Call-by-Value

```java
class NoSwapping {
    public static void main(String[] args) {
        String a = new String("Hello");
        String b = new String("World");
        System.out.println("String a = " + a + ", String b = " + b);
        swap(a, b);
        System.out.println("String a = " + a + ", String b = " + b); // no swapping!
    }
    public static void swap(String i, String j) {
        String temp = i;
        i = j;
        j = temp;
    }
}
```

```
String A = Hello, String B = World
String A = Hello, String B = World
```

Java: Swapping-Lösung mit Wrapper

```
class Swapping {
    public static void main(String[] args) {
        WrapperString a = new WrapperString("Hello");
        WrapperString b = new WrapperString("World");
        System.out.println("String a = " + a + ", String b = " + b);
        swap(a, b);
        System.out.println("String a = " + a + ", String b = " + b); // swapping!
    }
    public static void swap(WrapperString a, WrapperString b) {
        String temp = a.text;
        a.text = b.text;
        b.text = temp;
    }
}

public class WrapperString {
    public String text;
    public WrapperString(String text){this.text = text;}
    public String toString(){return text;}
}
```

```
String A = Hello, String B = World
String A = World, String B = Hello
```

Sie benötigen eine Wrapper-Klasse, um die beiden Strings zu vertauschen!

Arrays

- **Homogene Datentypen mit fester Länge**
- Arrays können 1-dimensional oder mehrdimensional sein
- Ein Array ist ein spezielles Objekt in Java, das mehrere Werte in einer einzigen Variablen speichert
  - z.B. `int[] myNum = {10, 20, 30, 40};`
- Ein Array ist ein spezieller Referenztyp in Java, der mehrere Funktionen bietet
  - z.B. die Array-Länge `System.out.println(myNum.length); // 4`

Looping in Arrays mit for und for-each

- Die klassiche `for`-loop mit index

```
for (int i = 0; i < myNum.length; i++)
    System.out.println(myNum[i]);
```

- ist äquivalent zu der `for-each`-Schleife:

```
for (int element : myNum)
    System.out.println(element);
```

- Java bietet einen Zugriff auf den Index über das `Iterator`-Interface und erlaubt eigene Implementierungen, wie es über ein Array iterieren soll!
  - Lernen wir noch kennen (siehe die Kapitel: Interfaces, Collections)

Keywords `break` and `continue`

- Keyword **`break`**
  - wird verwendet, um aus einer `switch-Anweisung` **herauszuspringen.**
  - wird verwendet, um aus einer Schleife zu springen (do, while, for).
- Keyword **`continue`**
  - **unterbricht nur eine Iteration** (in einer Schleife).
  - fährt mit der nächsten Iteration in der Schleife fort.
- Ergänzung: **`return`**
  - **Beendet die gesamte Methode** sofort und gibt (optional) einen Wert zurück.

```java
int sum = 0;
for (int i = 0; i < 10; i++) {
    if (i < 4) {
        sum += 1;
        continue;
    } else {
        sum += 10;
        break;
    }
}
System.out.println(sum); // 14
```

Array Klassen

- Java unterstützt **Array-Klassen** für
  - Primitive Datentypen (`bool, byte, int`,…)
  - Benutzerdefinierte Datentypen (`class`)
- [] gehören zum Klassennamen und legen die Anzahl der Dimensionen und die Array-Größe fest
  - z.B.: `int[2][5]` → Objektarray von je 2 Dimensionen mit 5 Einträgen
- Die Abmessungen und die Größe eines Arrays können nach der Erstellung nicht geändert werden
- `Person[] employee` kann eine Array-Referenz auf ein eindimensionales Array vom Typ `Person` haben

Array Klassen

- Die Arrays-Klasse enthält mehrere statische Methoden, die zum Füllen, Sortieren, Teilen, Suchen, Nichtabgleichen, Vergleichen, Kopieren usw. verwendet werden können.
- Beispiel: `parallelSort` (ein Multithread-Sortieralgorithmus) und `toString` (Ausgabe):

```
int intArr[] = { 10, 20, 15, 22, 35 };
Arrays.parallelSort(intArr);
System.out.println("Integer Array: " + Arrays.toString(intArr));
// Integer Array: [10, 15, 20, 22, 35]
```

Java Collections

- Können auch automatisch vergrößert und verkleinert werden, wenn Objekte hinzugefügt oder entfernt werden.
- Grundlegende **primitive Datentypen** können **nicht** gespeichert werden (zur Erinnerung: Autoboxing)
- Implementieren von **wiederverwendbaren Sammlungsdatenstrukturen**
  - Lists (ArrayList, LinkedList, Vector, and Stack)
  - Sets (Hashset, TreeSet)
  - Maps (HashMap, TreeMap)
  - Dictionary (Hashtable)
  - Queues, Deques

Java Recap Zusammengefasst

- Vorteile und Nachteile von Java
- Aufbau und Funktionsweise der Java Virtual Machine
- Automatisches Speichermanagement
- Aufbau von Klassen, Methoden, Variablen usw.
- Primitive und Referenztypen
- Typsicherheit in Java
- if/else, switch, Schleifen, break/continue/return
- Java ist immer Call-by-Value
- Methoden, Überladung
- Iteration, Array-Verhalten, Collections