# Java™ Crash Course

Lecturer: **Nataliia Stulova**
Teaching assistant: **Mohammadreza Hazirprasand**

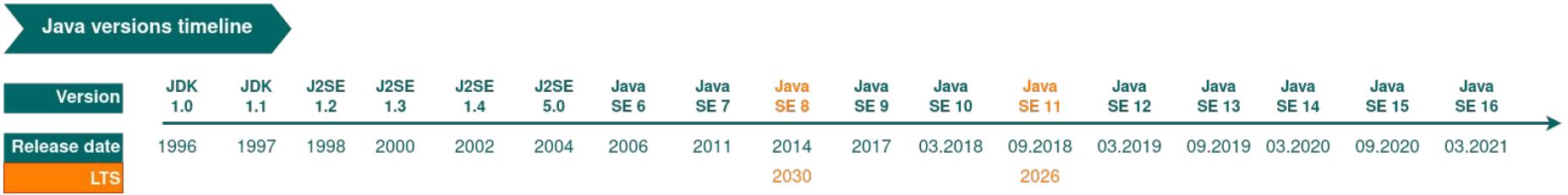
UNIVERSITÄT BERN

MASTER IN COMPUTER SCIENCE
OF THE UNIVERSITIES OF BERN, NEUCHÂTEL AND FRIBOURG

Software Skills Lab

# Part 1: Java ecosystem

# Java is...

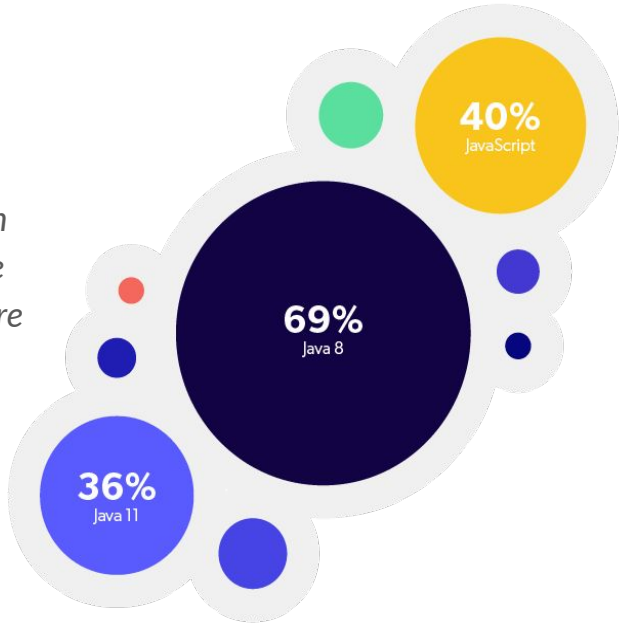



- a programming language
- an environment to run applications written in this language

# Why old(er) Java?

***Java 2021 Technology Report by JRebel:***

> *We asked developers what Java programming language they are using in their main application. Developers were able to select multiple answers if they are using more than one language, as this tends to be most common.*




Java 8 **69%**

JavaScript **40%**

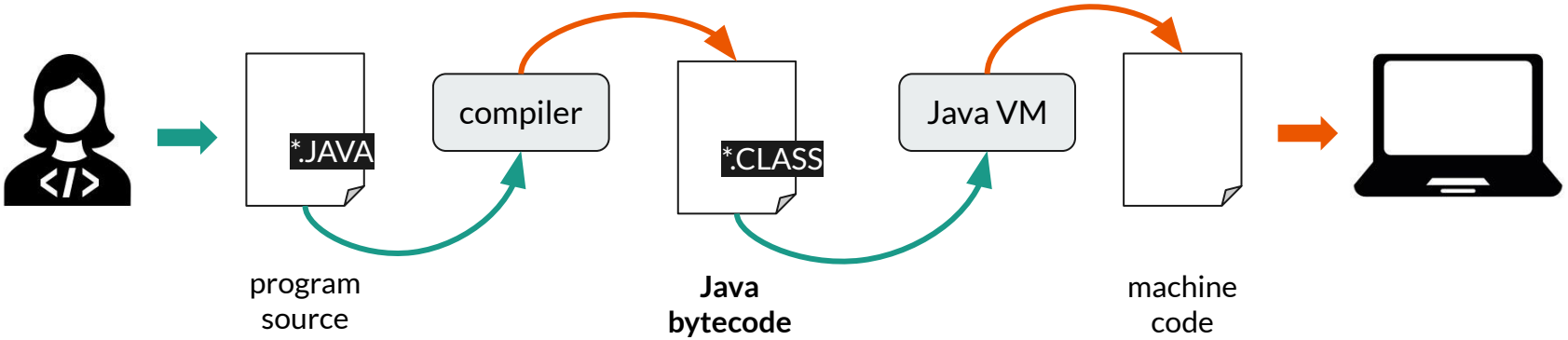
Java 11 **36%**

Java 12 or Newer **16%**

Java 7 or Older **15%**

https://www.jrebel.com/blog/2021-java-technology-report

# WORA: Write Once Run Anywhere




**Java bytecode**:

- intermediate representation interpreted by the Java Virtual Machine (JVM) *for a specific platform*
- does not depend on exact hardware architecture (= run anywhere)

# Java lingo

- **Java SE/EE/ME**: Java Standard/Enterprise/Micro **platform**
  - collections of tools to develop Java programs and the environment to run Java programs
- **JDK**: Java Development Kit
  - an implementation of one of the platforms (differ by sets of **tools**)
  - we will use some in this course: **java**, **javac**, **javadoc**, **jar**
- **JRE**: Java Runtime Environment
- **JVM**: Java Virtual Machine

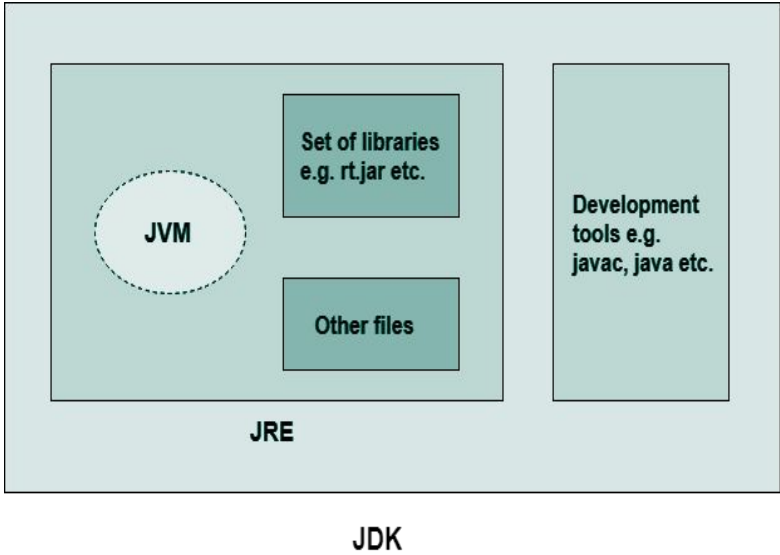
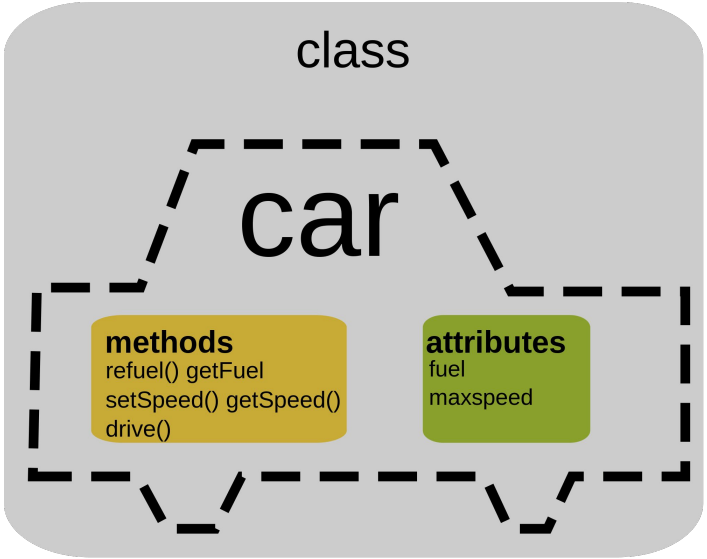



Image credits: https://www.javatpoint.com/difference-between-jdk-jre-and-jvm

# Part 2: Java basic syntax

# Java programming language

- **object-oriented**
  - (almost) everything is an object of a class
  - classes describe how the data is represented (via *attributes*) and manipulated (via *methods*)
- imperative
  - programmer specifies computational steps



# Hello, World!

everything is an object of a **class**

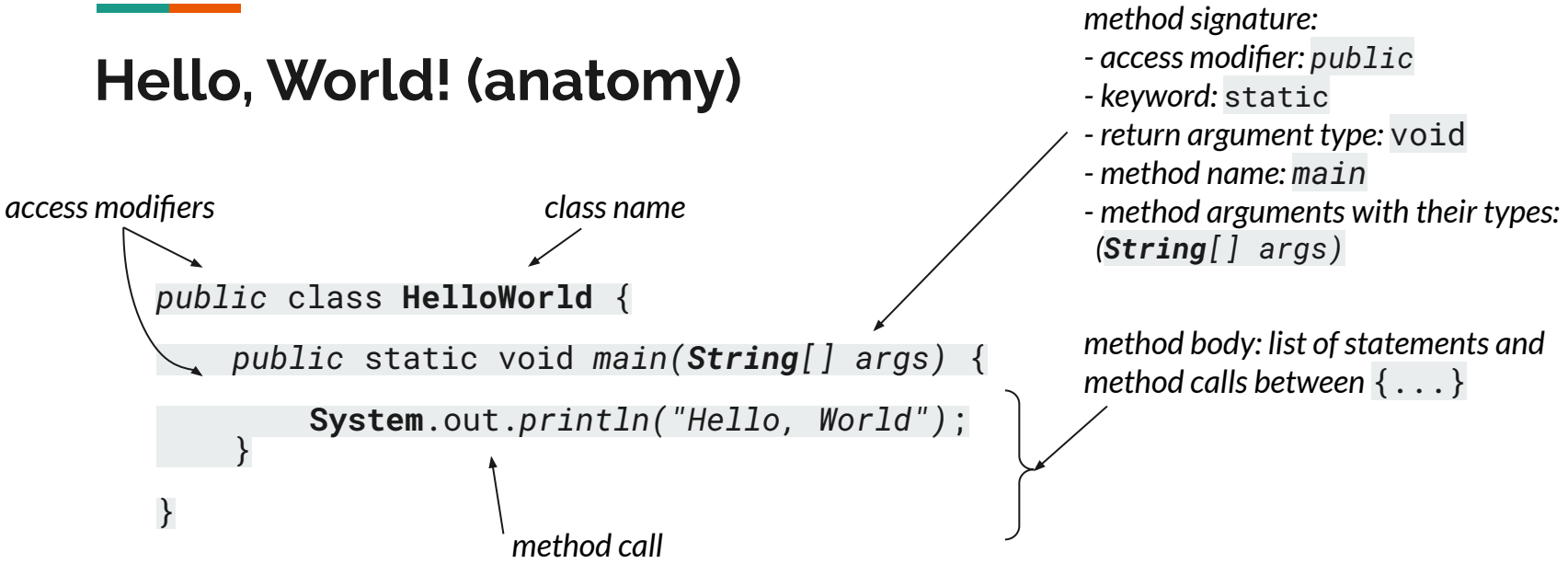
`main()` *method* - application entry point

```
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello, World");
    }
}
```

The `main()` is the starting point for JVM to start execution of a Java program.

Without the `main()` method, JVM will not execute the program

# Hello, World! (anatomy)




*a static method can be invoked without the need for creating an instance of a class

# Primitive and reference types

- Primitive data types (hold values)
  - `byte` < `short` < `char` < `int` < `long` < `float` < `double`
  - `boolean`
- All other types are reference types (hold references to objects)
- `null` - special reference, does not refer to anything, “empty” reference

# Operators

For primitive types:

- Assignment: `=`
- Arithmetic: `+`, `-`, `*`, `/`, `%` (integer division)
- Comparison: `>`, `>=`, `<`, `<=`, `==` (equality) , `!=` (inequality)
- Conditional: `&&` (AND), `||` (OR), `!` (NOT)

Different in reference types:

- Reference equality: `==`
- Contents equality: `.equals(...)`

Operators can be **overloaded** - given new meaning - in reference types.

For example, `+` is used for concatenation in Strings:

```
String s1 = “Hel” + “lo”;
String s2 = “Hello”;

System.out.println(
      s1.equals(s2));
```

# Conditionals

Executing different code depending on some logical (=`boolean`-valued) conditions:

```
if (CONDITION1) {
    ...
} else if (CONDITION2) {
    ...
} else {
    ...
}
```

Executing different code depending on fixed values of a variable:

```
switch (VARIABLE) {
    case VALUE1:
        ...
        break;
    case VALUE2:
        ...
        break;
    default:
        ...
        break;
}
```

# Loops

Java has 3 kinds of loops...

- `for (i = 0; i < N; i++) { code }`
- `while (condition) { code }`
- `do {code} while (condition)`

...and two special loop statements

- `continue` - start next loop iteration
- `break` - exit the loop

prints numbers 0-4

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

prints number 1

```
for (int i = 0; i < 5; i++) {

    if (i % 2 == 0) continue;

    if (i == 3) break;

    System.out.println(i);
}
```

skip even numbers

stops reaching 3

# IO: Input and Output

*Java has 3 streams called System.in, System.out, and System.err which are commonly used to provide input to, and output from Java applications.*

```
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello, World");
    }

}
```

Program input:
- CLI **arguments** (here)
- **`System.in`**

Program output:
- **`System.out`** (here)
- **`System.err`**

# Stream IO

**Interactivity: reading from input stream:**

The code on the right:

1. **creates** a `Scanner` object
2. **uses** it to read a `String` and an `int`
3. prints to the output stream, and
4. **closes** the `Scanner` object because there is no more input to read

**Hint:** always close the input stream!

```
import java.util.Scanner;

...
    Scanner scanner = new Scanner(System.in);

    String myString = scanner.next();
    int myInt = scanner.nextInt();
    System.out.println("myString is: " + myString);
    System.out.println("myInt is: " + myInt);

    scanner.close();
```

# File IO

## Reading a line from a text file

```
File myFile = new File("PATH/test.txt");
Scanner myReader = new Scanner(myFile);

String textLine = myReader.nextLine();
System.out.println(textLine);

myReader.close();
```

## Writing a line to a text file

```
FileWriter myWriter = new FileWriter("PATH/test.txt");

myWriter.write("Hello, world!");

myWriter.close();
```

# Exceptions

If a method does not handle an exception, the method must declare it using the throws keyword at the end of a method's signature.

```
public void myMethod() throws IOException{
      ...
      throw new IOException();
}
```

```
public void myMethod() throws IOException{
      ...
     myOtherMethod();
}
```

*this method actually throws an exception*

Handling **unexpected behavior**:

```
try {
  ...code that might throw an exception

} catch (ExceptionType1 e1) {
  ...process exception

} catch (ExceptionType2 e2) {
  ...

} finally {
  ...code that always executes.
}
```

# Comments

```
/** This is a class-level doc comment.
 */
public class HelloWorld {

    /** This is a method-level doc comment. This is free-text comment part.
     *  @param args This is tagged comment part
     */
    public static void main(String[] args) {

        // this is an inline comment
        System.out.println("Hello, World");
    }
      /* this is a
         multi-line block comment */
}
```

# Part 3: Java applications

# Compiling and running

Java code is usually organized as a **project**.

Project file hierarchy:

- project (collection of packages)
  - package (collection of classes)
    - class

3 options to produce an executable program:

- **CLI:** text editor + `java`, `javac`, jar
- **IDE**: Eclipse, NetBeans, IntelliJ, VisualStudio Code,...
- **Build systems**: Maven, Gradle

(or a combination of!)

# CLI compilation and execution

Make a folder with the following structure:

- your-program-name
  - Hello.java
  - other *.java files
  - MANIFEST.MF

The manifest file should specify main class:

```
Main-Class: Hello
```

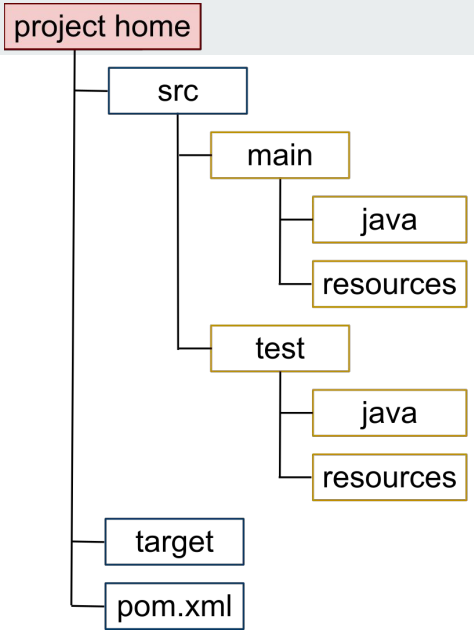
**Option 1:**

```
$ javac *.java
$ jar cfm hello.jar MANIFEST.MF *.class
$ java -jar hello.jar
```

**Option 2:**

```
$ javac *.java
$ java Hello
```

# Java Maven projects



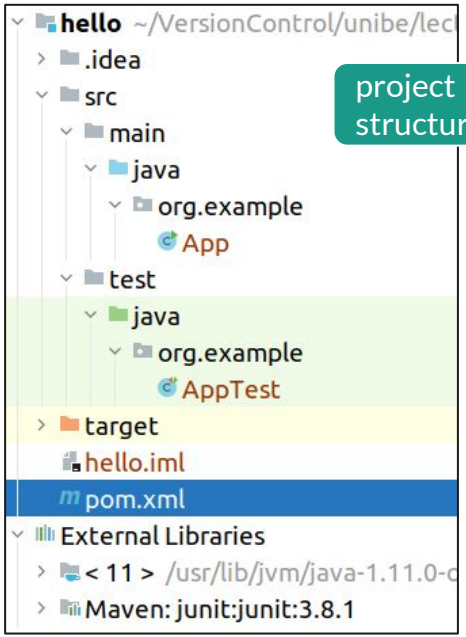

Maven is a build automation tool for managing:

- project structure
- software build cycle stages (compilation, testing,...)
- dependency management (external libraries, other projects,...)

Maven dynamically downloads Java libraries and Maven plugins from one or more repositories.

A Project Object Model (POM) file `pom.xml` provides all the configuration for a single project.

# Maven™ POM file structure




project structure

```
<modelVersion>4.0.0</modelVersion>

<!--company name-->
<groupId>org.example</groupId>
<!--application name-->
<artifactId>hello</artifactId>
<version>1.0-SNAPSHOT</version>
```

1: POM header

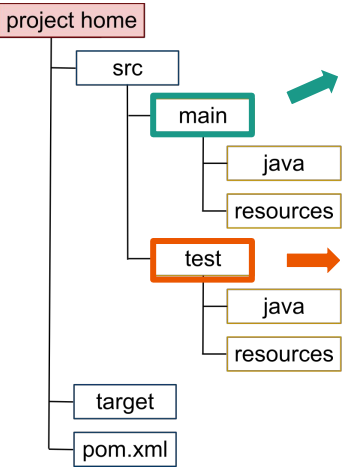
```
<dependencies>
 <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>3.8.1</version>
   <scope>test</scope>
 </dependency>
</dependencies>
```

2: libraries

```
<build>
 <plugins>
   <plugin>
     <groupId>
       org.apache.maven.plugins
     </groupId>
     <artifactId>
       maven-compiler-plugin
     </artifactId>
     <version>3.6.2</version>
     <configuration>
       <!--Java language version-->
       <source>11</source>
       <target>11</target>
     </configuration>
   </plugin>
 </plugins>
</build>
```

3: build setup

# Unit testing with JUnit framework




```
public class MyUnit {
    public String concatenate(String one, String two){
        return one + two;
    }
}
```

*unit of code under test: method* `concatenate()`

```
public class MyUnitTest {
      @Test
      public void testConcatenate() {

          MyUnit myUnit = new MyUnit();
          String result = myUnit.concatenate("one", "two");

          assertEquals("onetwo", result);

      }
}
```

*unit test for the method* `concatenate()`

# Coding conventions: which?



Code Conventions for the Java™ Programming Language

Google Java Style Guide

- most IDEs have support for project-level style set up
- styles can differ between projects, so **agree with collaborators**

# Part 4: practice

# Exercise 1: basic Java application

- Set up Java on your machine and an IDE: either IntelliJ (recommended) or Eclipse
- Write the simplest program: Hello, world!
  - Set up a simple Java application project and compile it through the CLI or within the IDE
  - Produce an executable JAR and execute it
- Expand it to the unit test example program: string concatenation
  - A printer program that: reads a number N from System.in, if it is even prints N characters '-' to the standard output stream, if it is odd - prints N characters '=' to the standard error stream.
  - Demonstrate use of stream IO, conditionals, operators, loops, comments, [optional: exceptions]

# Exercise 2: Java Maven application

- Set up a maven project
- Write a program that copies text files: reads a line from one file and writes it to another file
  - Demonstrate use of file IO, conditionals, operators, loops, comments, exceptions, unit tests

# Further resources on Java

# General Java tutorials

**Oracle Java tutorial**

**Lesson: Classes and Objects (The Java™ Tutorials > Learning the Java Language)**

**Online crash courses**

Udemy: Java Beginners Program - A crash course

University of California, Berkeley: A Java Crash Course

**Other tutorials**

Java Tutorial | Learn Java Programming - javatpoint

TutorialsPoint Java Tutorial

W3Schools Java Tutorial

HowToDoInJava: Learn Java, Pyhton, Spring, Hibernate

# Thematic resources

- Code examples collection of basic Java concepts
- StackOverflow - programming community Question/Answer website
- Maven in 5 minutes, or in a brief example
- Unit testing with JUnit:
  - JUnit - Test Framework
  - Unit Testing with JUnit 5 - Tutorial