Java Fundamentals

Section 1 - Introduction to Java

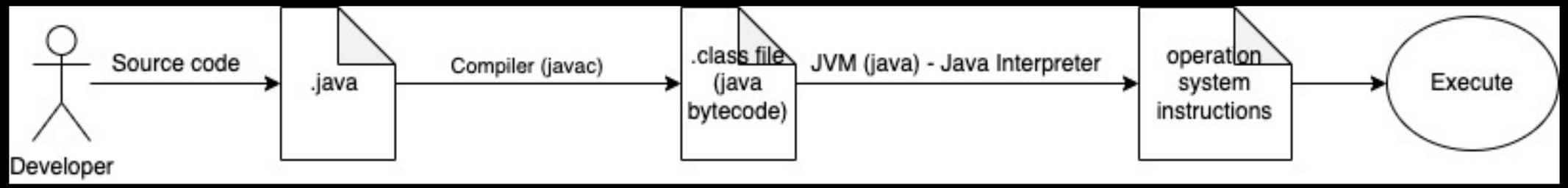
Topics

- What is Java ?
- Prerequisites
- IntelliJ IDEA Community IDE
- Object Oriented Programming
- Primitives
- Sytax
- Packages, Classes, Objects
- Pass by value or pass by reference?
- Access modifiers
- Beginners mistakes
- Coding standards
- Wrapper classes
- Arrays
- Strings
- Static and final keywords
- Code examples: https://github.com/diana-stoica-ub/java-fundamentals-may-2022

What is Java?

What is Java?

- Platform-independent, **portable** programming language: write code once and run it on almost any computing platform
- Object Oriented Programming (**OOP**) Language
- Multi purpose - it can be used for enterprise software, mobile applications, web apps, desktop apps, etc
- multithreaded language
- automatic memory management



What is Java?

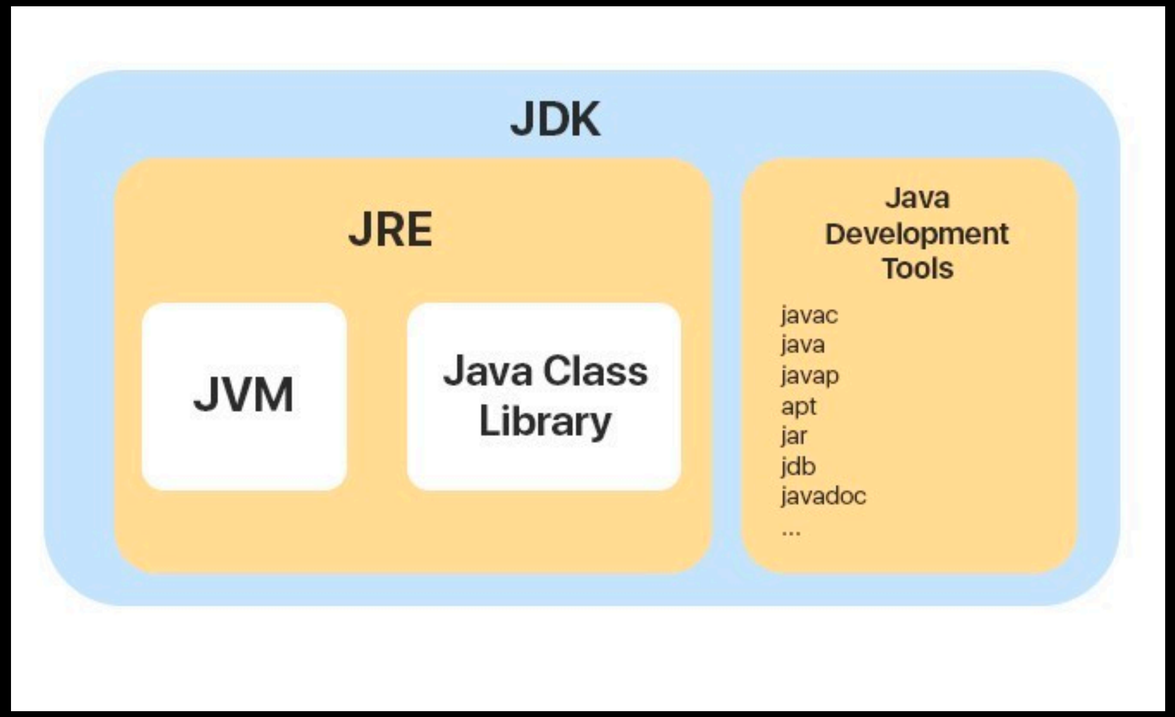
Structure of a Java application

- **Primitives** - simple data types
- **Classes** - complex data types, abstractions over real life concepts
- **Object** - instance of a class; memory is allocated
- Classes are grouped into **packages**
- **Main method** - starting point of the Java application
- **null** - absence of reference, unallocated variable
- **import** - if we need to use classes from other packages (except from package java.lang which is imported by default)

What is Java?

JVM, JRE, JDK

- **JVM** = Java Virtual Machine
  - Converts bytecode to machine code
  - Memory management, garbage collector
- **JRE** = Java Runtime Environment
  - Contains everything needed for running Java applications (including the JVM)
  - Java libraries
- **JDK** = Java Development Kit
  - Contains everything needed for developing and debugging Java applications (including JRE)
  - tools, executables, and binaries required to compile, debug and execute a Java program - eg: interpreter (java), compiler (javac), documentation generator (javadoc), archiver (jar), etc



JDK releases

- Since Java 9, a new JDK version comes out every 6 months (March and September) with each release supported for their half-year lifespan
- Each three years, there will be a LTS (Long Term Support) version: 8, 11 and 17
- There are multiple JDK implementations:
  - By Oracle: Open JDK, Oracle JDK
  - By other vendors like Microsoft, Azul, Amazon, etc.

Prerequisites

Prerequisites

JDK & IDE

- In this course we are going to use Java 17 (Oracle OpenJDK 17)
- Download sources:
  - https://jdk.java.net/java-se-ri/17
  - https://www.oracle.com/java/technologies/downloads/#jdk17-linux
- IntelliJ IDEA Community Edition:
  - https://www.jetbrains.com/idea/download
  - Note: for enterprise development, IntelliJ Ultimate Edition is needed (it requires a license)

IntelliJ IDEA

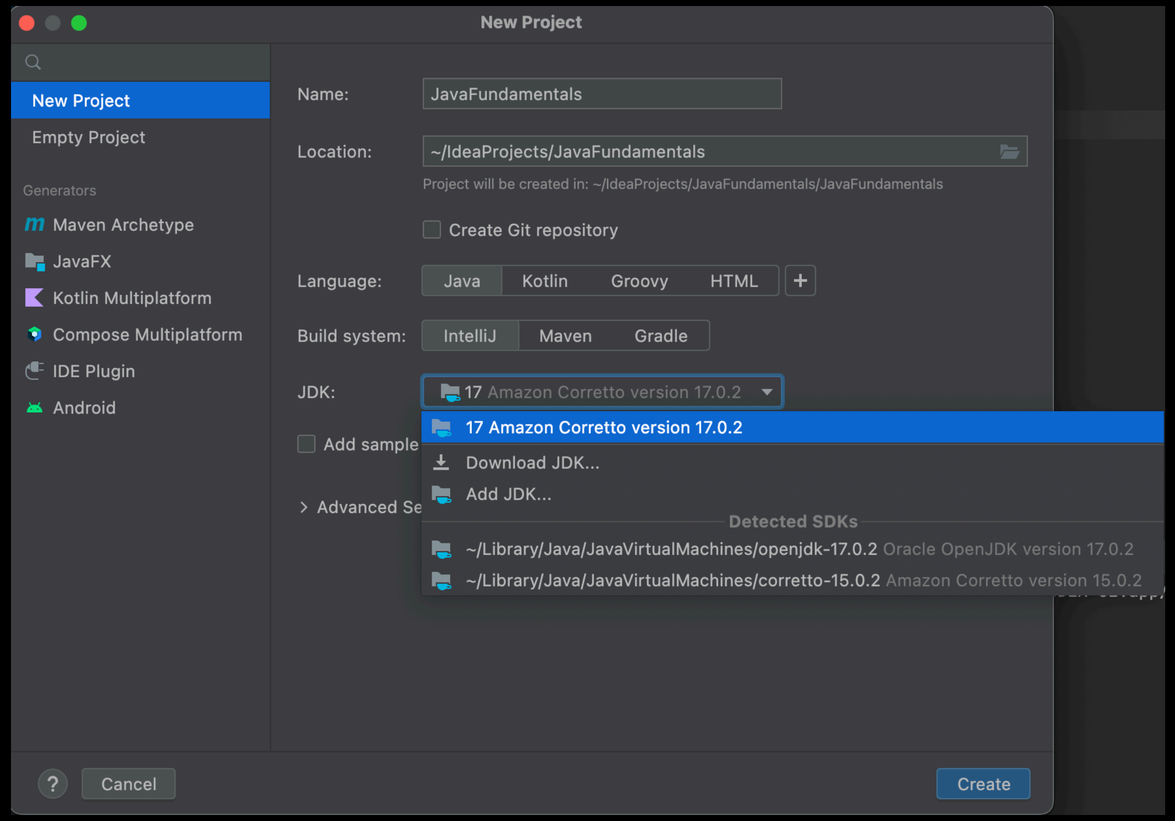
Create a new project

- Create a new project: File -> New -> …
  - Project - creates a new project from scratch
  - Project from Existing Sources - if we have the code already on our local machine and we want to create an IntelliJ Project from it
  - Project from Version Control - if we want to clone a project from a VC System (eg: Git)

IntelliJ IDEA

Create a new project

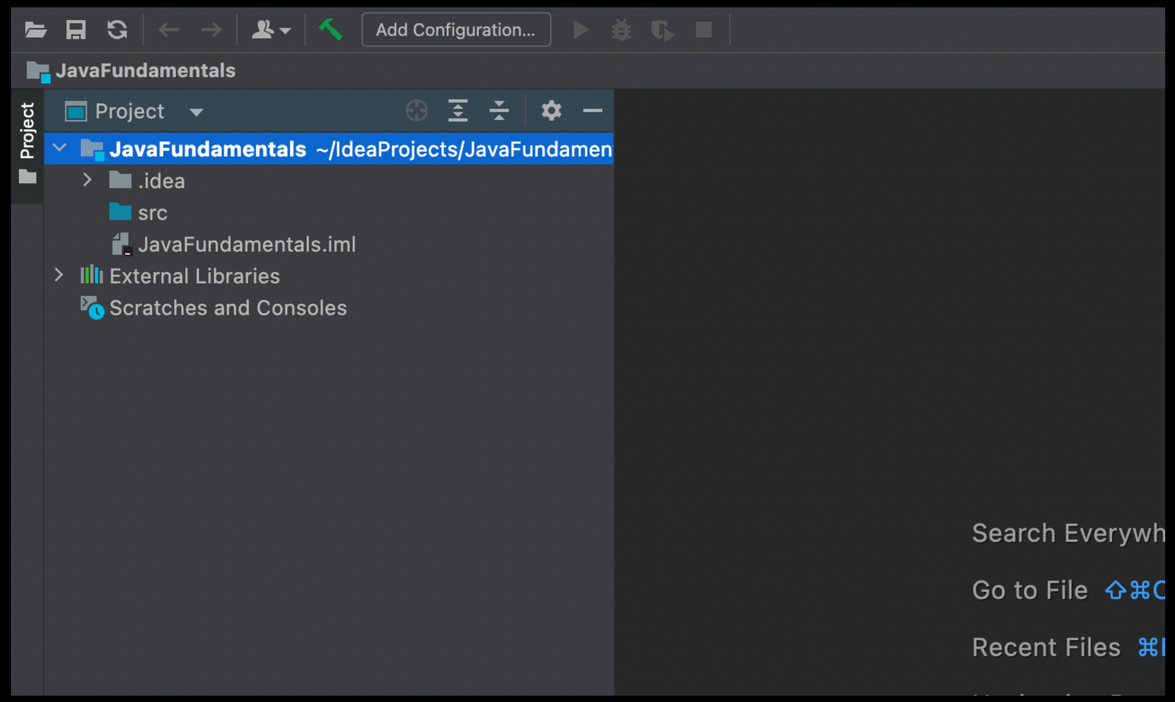
- Name: name of the project
- Location: local path where the project will be saved
- JDK:
  - select the JDK that you want to use
  - Add a JDK from local source
  - Download a JDK



IntelliJ IDEA

Create a new project

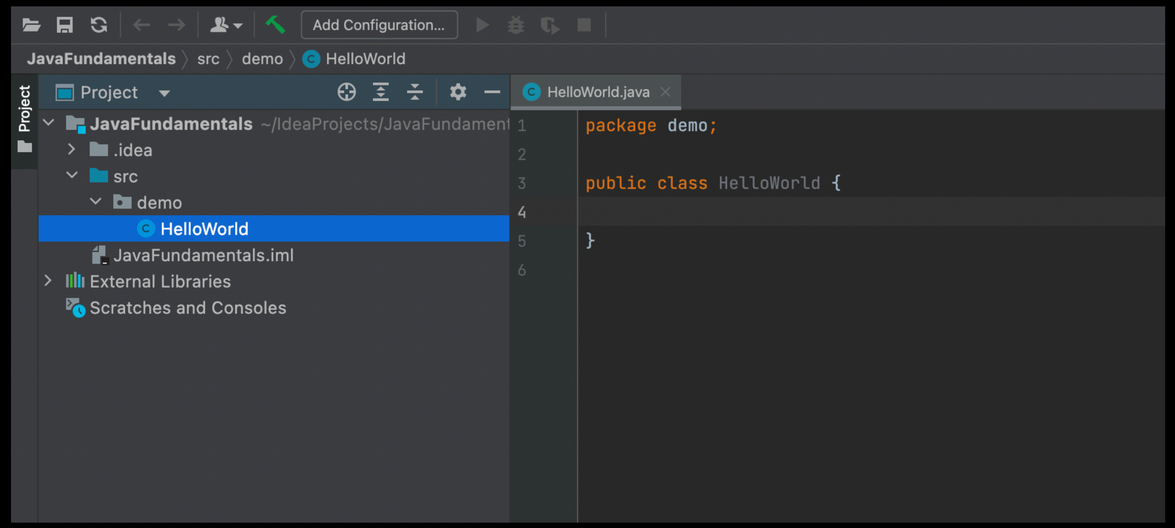
- /.idea; *.iml - metadata files & configurations (if you are using a VCS system, do NOT commit this!)
- /src - the source files will be added here
- External Libraries - here the imported libraries can be examined



IntelliJ IDEA

Creating the first Java app

- Right click on /src folder -> Create new -> package
  - Name: demo
- Right click on the new /demo package -> Create new -> class
  - Name: HelloWorld



IntelliJ IDEA

Creating the first Java app

- Each Java app must have a main method - entry point of the application
- *public static void main(String[] args)*
- Write a simple main method that displays a message into the console

HelloWorld.java

```java
package demo;

public class HelloWorld {

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

}
```

IntelliJ IDEA

Running the first Java app

- There are multiple ways to run a Java app from IntelliJ:
  - Right click on the main method / main class and select Run HelloWorld.main()
  - Click on the green triangle and select Run HelloWorld.main()
- We can run an application in **debug** mode by selecting Debug HelloWorld.main() instead

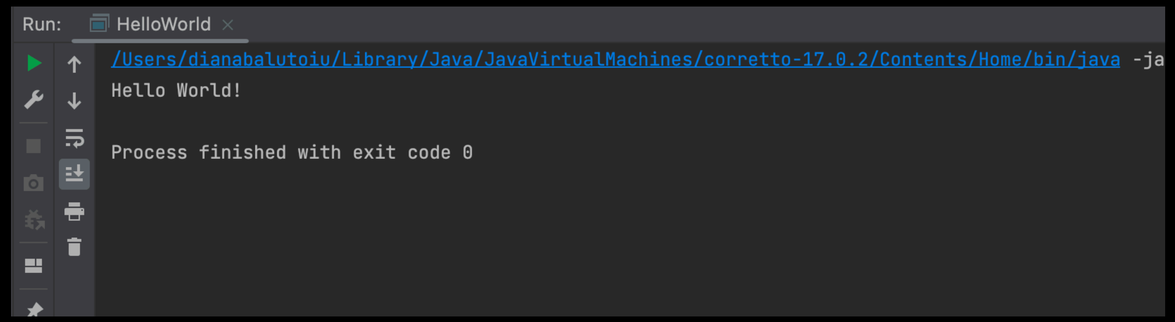
HelloWorld.java

```java
package demo;

public class HelloWorld {

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

}
```

IntelliJ IDEA

Running the first Java app

- The message will be displayed in the **console**
- Exit code 0 means that the program ran successfully
- If something wrong would have happened, an error / exception would have been displayed and status code would not have been 0.

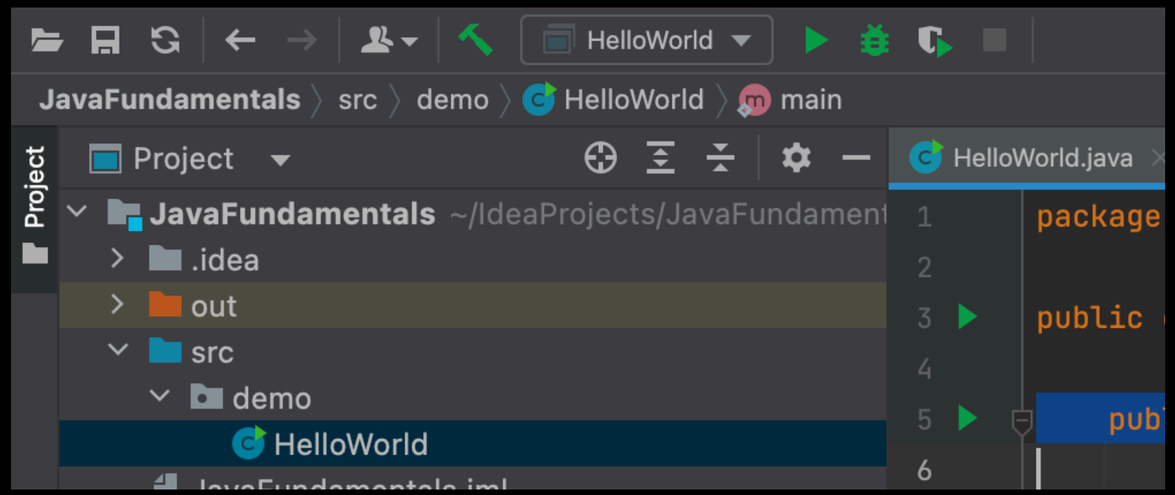


IntelliJ IDEA

Running the first Java app

- After we first run an application, IntelliJ will create a '**Run Configuration**' for it; so now we can run it from the Toolbar as well (View -> Appearance -> Toolbar in case it's hidden)

- If we click on the Run Configuration -> Edit Configuration, we can modify it (eg: adding program arguments or environment variables)



IntelliJ IDEA

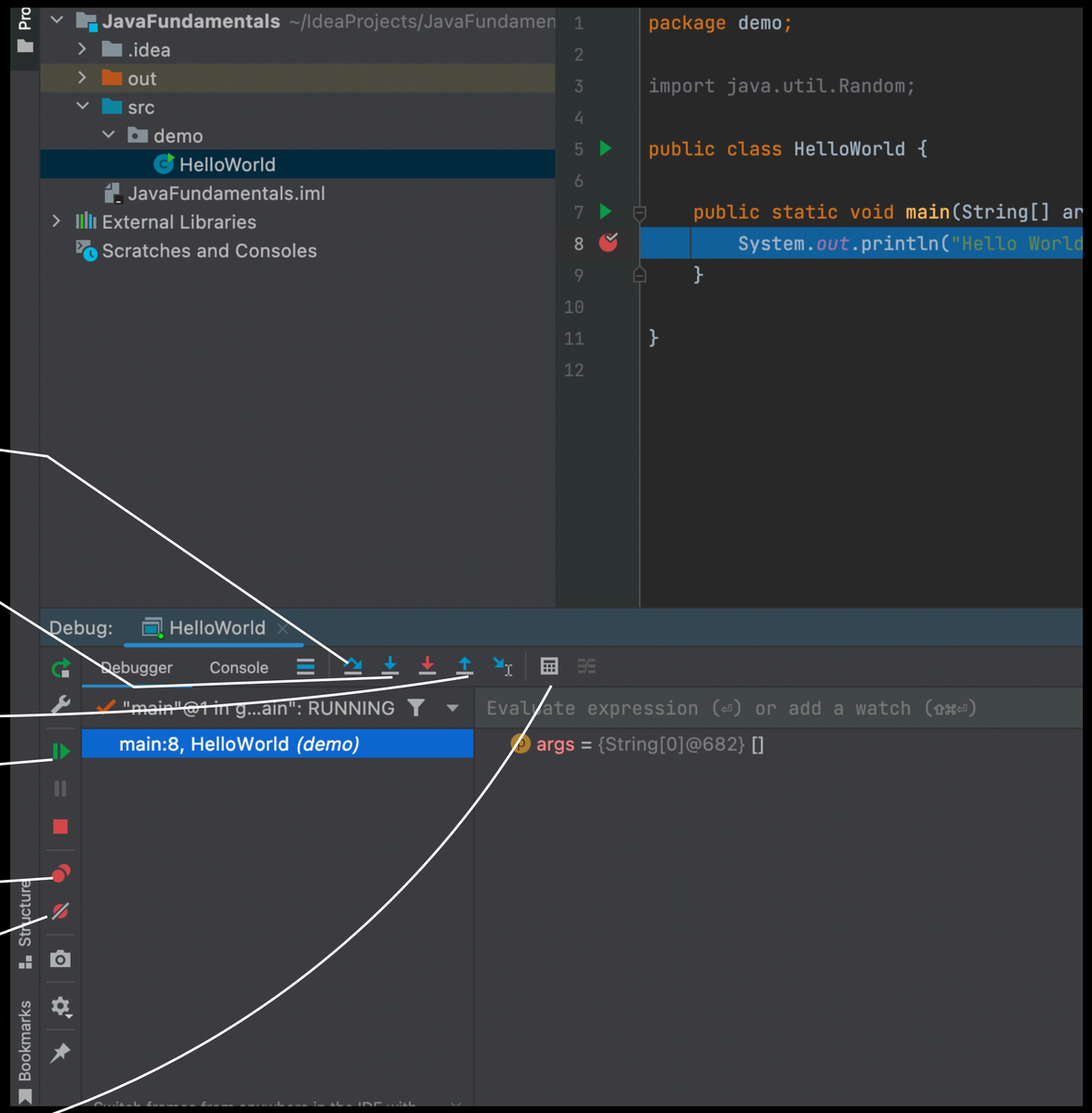
Debugging

- **Breakpoint**: a signal that tells the debugger to pause execution of your program at a certain point in the code
- When in a breakpoint, you can inspect variables values, execute methods
- To add a breakpoint, click near the line number

IntelliJ IDEA

Debugging

- When the execution is suspended in the breakpoint:
  - Step over (F8): jumps to the next line
  - Step into (F7): jumps into the called method (eg: into ‘println() method)
  - Step out (Shift + F8): jumps out of the current method into the caller one (eg: from println() back to main)
  - Resume program (F9): resume the execution
- Other actions:
  - View breakpoints
  - Mute all breakpoints: temporarily disable all breakpoints
  - Evaluate expression



IntelliJ IDEA

Cheatsheet

- Alt + Enter : fix anything in the given context(if we have an error or warning, it will give fixing suggestions; intention actions)
- Ctrl + Space : Basic Code Completion
- Ctrl + Click : Go to method / class definition
- Ctrl + N : Search Class by name
- Ctrl + Shift + N : Search File by name
- Ctrl + Shift + F : Search word everywhere
- Ctrl + P (on method): display accepted parameter types
- Alt + F7 : Find usages
- Alt + F8 : Evaluate Expression (in debug mode)
- For more shortcuts / updates, See *Preferences -> Keymap*

Java - object oriented programming language

What is Java?

Object oriented programming language

- OOP = programming paradigm that uses **objects** and **interactions** between them in order to model the architecture of the application
- **Focus on objects** that developers want to manipulate rather than the logic required to manipulate them (functions, procedures)
- OOP objects should model **real life objects** or concepts (eg: Student, Employee, Department)
- First step = identify the objects that we will need to model -> **data modelling**

What is Java?

Object oriented programming language - the principles

- **Encapsulation** – the state and functionalities of a class are hidden from others (private); access only via non-private methods
- **Abstraction** – hiding implementation details, only reveal operations that are relevant for other classes
- **Inheritance** – a class can extend another class and it inherits its state and functionality
- **Polymorphism** – ability of an object to take many forms (e.g.: if A extends B, then an instance of class A is an A, but also a B, and an Object)

Object oriented programming

Keywords

- **Class**: user defined data-type that models the real life object/ concept; blueprint, abstraction (eg: Department)
- **Object**: instance of a class, memory allocated (eg: The HR Department); class vs object => concept vs real life object (eg: “chiar” vs the actual chair that you’re s
- **Method**: functions that define actions that an object can make (eg: createDepartment)
- **Attribute**: fields of a class, properties (eg: department’s name)

Object oriented programming

Why?

- **Readability**: easier to use and read, since most of the complexity is hidden
- **Extensibility**: well-suited for programs that are large, complex and actively updated/maintained
- **Collaborative development**: easy to split the work between multiple team members and track the progress
- **Reusability**: classes can be reused for multiple problems
- **Testability**: code is easier to test

Object oriented programming

Why not?

- OOP overemphasises the data component of software development and does not focus enough on logic or algorithms.
- code can be more complicated to write
- code can take longer to compile.

Primitives

Primitives

What are primitives?

- OOP paradigm says that anything should be an object
- However, for performance reasons, Java supports basic types named **primitives**
- Because it has primitives, Java is not a pure OOP language
- Primitives are not classes, not objects
- Primitives are initialised with default values if they are not set (0 for numeric types, false for boolean)
- **void** is not a primitive; it's just a keyword that is used for methods that return **nothing**
- More on primitives here

Primitive Type Keyword

Type	Size in bytes	Range	Default Value
byte	1 byte	-128 to 127	0
short	2 bytes	-32,768 to 32,767	0
int	4 bytes	-2,147,483,648 to 2,147,483, 647	0
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0
float	4 bytes	approximately ±3.40282347E+38F (6-7 significant decimal digits) Java implements IEEE 754 standard	0.0f
double	8 bytes	approximately ±1.79769313486231570E+308 (15 significant decimal digits)	0.0d
char	2 bytes	0 to 65,536 (unsigned)	'\u0000'
boolean	Not precisely defined*	true or false	false

Syntax

Syntax

Operators - arithmetic

- Assuming we have two variables declared:

```
int a = 10;
int b = 3;
```

+	Addition	a + b =>13
-	Subtraction	a - b => 7
*	Multiplication	a * b => 30
/	Division	a / b => 3
%	Modulus	a % b => 1
++	Increment	a++ => 11 ++a => 11
- -	Decrement	a - - => 9 - - a => 9

Syntax

Operators - relational

- Assuming we have two variables declared:

```
int a = 10;
int b = 3;
```

==	Equal to	a == b => false
<	Less than	a < b => false
<=	Less or equal to	a <= b => false
>	Greater than	a > b => true
>=	Greater or equal to	a >= b => true
!=	Not Equal to	a != b => true

Syntax

Operators - logical

- Assuming we have two variables declared:

```
boolean a = true;
boolean b = false;
```

&&	Logical and	a && b => false
||	Logical or	a || b => true
!	Logical not	!a => false !b => true

Syntax

Operators - other operators

- Assignment operators:
  - = (c = a + b will assign value of a + b into c)
  - += ; -=; /=; %=; *= (aritmetic operation and assignment: b+=a is equivalent to b = b + a)
- Bitwise operators: &; |; ^; >>; <<; >>>
- Ternary operator

```
int a = 10;
int b = 3;

int max = (a > b) ? a : b;
```

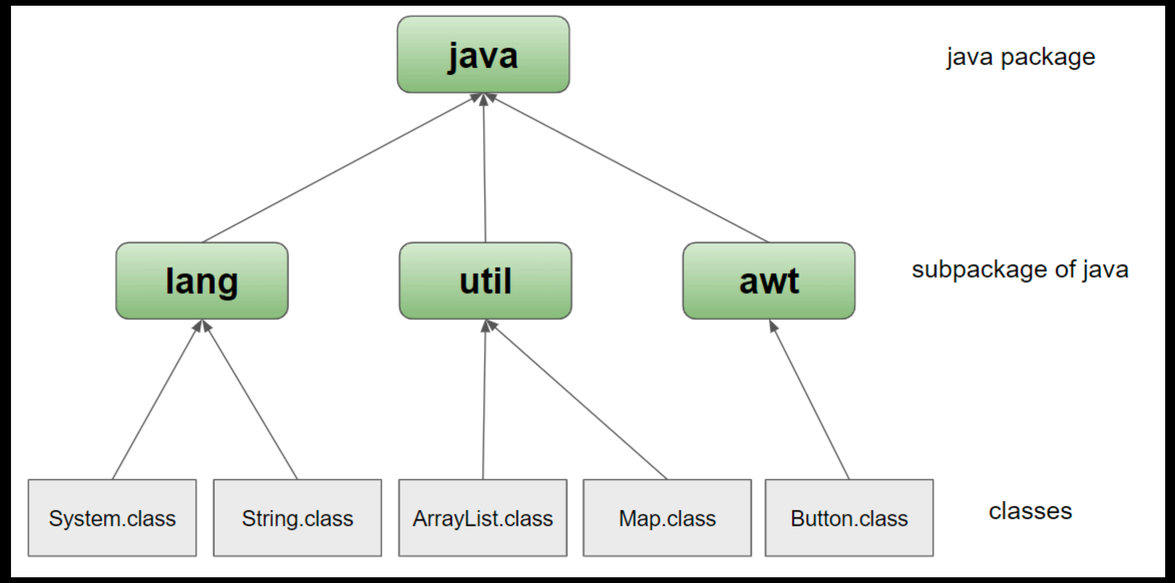
Syntax

- More on syntax here

Packages, Classes, Objects

Packages

- A java application is split in logical groups named packages
- Keyword: **package**
- A package can contain source files (*.java files, which contain the classes) and other packages
- Read more here and here



Packages

- You can create a class and don’t place it in a package (**default package** or unnamed package)
- However, using the default package is not recommended because:
  - We lose the advantage of having a structure, we can’t have sub packages.
  - We can’t import the classes from the default package into other packages
  - Lack of encapsulation (some of the access modifiers will be meaningless)

Packages

Naming rules & conventions

- **Rules**:
  - Characters allowed: letters, “$”, “_” , numbers(not at the start of the name)
  - Name cannot be a keyword (eg: class, package)
- **Conventions**
  - Lower case (eg: com.amazon.service)
  - Usually nouns
  - Companies use their reversed Internet domain name to begin their package names (eg: com.amazon as a root package for the amazon.com website app)
- Packages in Java correspond with a directory structure (com.amazon.service package => /com/amazon/service directory structure)

Packages

Using packages

- **package statement**: the very first line of code in a file.
- In order to use a class from a package in another package, we need to use **import statements**
- Exception: **java.lang** doesn't need to be imported
- We can import classes from packages defined by us, from JDK libraries (eg: java.util package) and from external libraries
- We can import the whole content of the package (*), just one class, or just a static member

```java
package demo;

import service.*;

import java.awt.*;
import java.util.Random;
import java.util.function.Function;

import static constants.Constants.CONSTANT;

public class HelloWorld {
```

Classes

- Keyword: **class**
- Keyword: **this** - references the current object
- Complex data type created by the user or already existing (eg: from the JDK class library, imported from other libraries)
- A class can contain:
  - **Attributes** (also referred by members, fields, properties) : data, they define the state of the object; they can be primitives or other class types
  - **Methods**: functions, they alter the state of the object
  - **Constructors**: special methods that are used to create an instance (object) of the class
- More about classes and objects here and here

Classes

Class vs object

- Class = abstraction, a concept (eg: Employee)
- Object = instance of a class; an actual representation of a class (eg: the Employee named John)
- When we instantiate (create) an object, memory is being allocated for it
- Keyword: **new**
- An object is being instantiated by calling the class constructor
- If we declare a variable of a class type, until instantiation, its value will be **null**

Classes

Naming rules and conventions

- **Rules**:
  - Characters allowed: letters, “$”, “_” , numbers(not at the start of the name)
  - Name cannot be a keyword (eg: class, package)
- **Conventions**:
  - Camel case: HelloWorld
  - Usually nouns: Employee, DepartmentCreator

Classes

Class vs file

- Usually, the name of the file containing the class will be the same as the name of the class itself (eg: HelloWorld class -> HelloWorld.java file) -> considered good practice
- However, we can have multiple classes in the same file, as long as:
  - We have at most one public class in the same file
  - The name of the file should be the same as the public class (if it exists)

Classes

Constructors

- A class can have multiple constructors constructors (**overloading**)
- They are invoked to create objects from the class blueprint
- they use the name of the class and have no return type
- If we don’t define a constructor - the **default constructor** will be created automatically (a constructor with no parameters); once we define a constructor, the default one will not be available anymore.

Classes

Enum

- Keyword: **enum**
- Special kind of class that can map a series of constants

HelloWorld.java | Colour.java

```java
package constants;

public enum Colour {
    GREEN,
    BLUE,
    BROWN
}
```

Access modifiers

Access modifiers

- Any class, method, attribute has an access modifier
- They are used to restrict the access to the entity from other classes:
  - **public** - access from external classes
  - **private** - limits the access to the current class
    - Can be used for classes, but only if they are inner classes
  - **protected** - limits the access to the current class and to child classes (classes that extend the current class)
    - Can be used for classes, but only if they are inner classes
  - **default** (no access modifier specified) - limits the access to the current package

Pass by value or pass by reference?

Pass by value or pass by reference?

- In Java, parameters are **passed by value**
- However, when the parameter is an object, it’s value is actually the **reference**

Beginners mistakes

Beginners mistakes

- When we start learning a new programming language, it is natural to make mistakes. This list should help you debug some of the most common issues that occur when starting to work with Java.

Beginners mistakes

- Writing instructions outside of methods or classes
- Writing methods or member fields outside the class definition

HelloWorld.java | Colour.java

```java
package demo;

System.out.println("This will not work");

public class HelloWorld {

    System.out.println("This will not work");

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

}
String s = "This will not work";
System.out.println("This will not work");
```

Beginners mistakes

- Pay attention to the indentation, always have the block accolades correctly closed
- Even though for a single instruction accolades are not required, they are recommended, because leaving them out might cause unintended bugs
  - Some IDEs will signal this issue - always pay attention to IDE warnings and errors

```
//code will not work as intended if Dev2 will not realise that he should add accolades {}
if (isSuccess)
    System.out.println("Dev 1: Writes this line of code");
    System.out.println("Dev 2: Writes this line of code");
}
```

Suspicious indentation after 'if' statement

Beginners mistakes

- Using “=“ instead of “==“ in if blocks/while blocks
  - However, some IDEs will signal this issue
- Comparing objects with “==“ instead of equals

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

    int a = 1;
    if (a = 2) {
        System.out.println("Don't confuse = and ==");
    }
}
```

Beginners mistakes

- Accessing a field of an object that was not yet instantiated - this will cause a NullPointerException
- Sometimes the IDE might be able to catch this issue and display a warning

```
Employee employee;
employee.name = "John";
```

Variable 'employee' might not have been initialized

Initialize variable 'employee' ⌥⇧⏎ More actions... ⌥⏎

Employee employee

JavaFundamentals

Coding standards

Coding standards

- Respect the Java coding style
- Naming: respect the naming conventions; give intuitive names to classes, variables, methods, etc (**explicit is better than implicit**)
- Don’t duplicate the code - if you need to reuse it, that means that it should be extracted into a separate method / class!
- Classes should have a single responsibility. Methods should be short and they should do only one thing
- Be consistent across the project

Wrapper classes

Wrapper classes

What are wrappers?

- Special classes that encapsulate primitive Java types (part of the **java.lang** package).
- Each Java primitive has a corresponding wrapper:
  - boolean : Boolean
  - byte : Byte
  - short : Short
  - char : Character
  - int: Integer
  - long : Long
  - float: Float
  - double: Double

Wrapper classes
Why?

- Generic classes don’t work with primitives
  - Eg: Collections
- Primitives can’t represent the absence of a value (null)
  - Sometimes, we need to differentiate the absence of a value from the “0” value

Wrapper classes

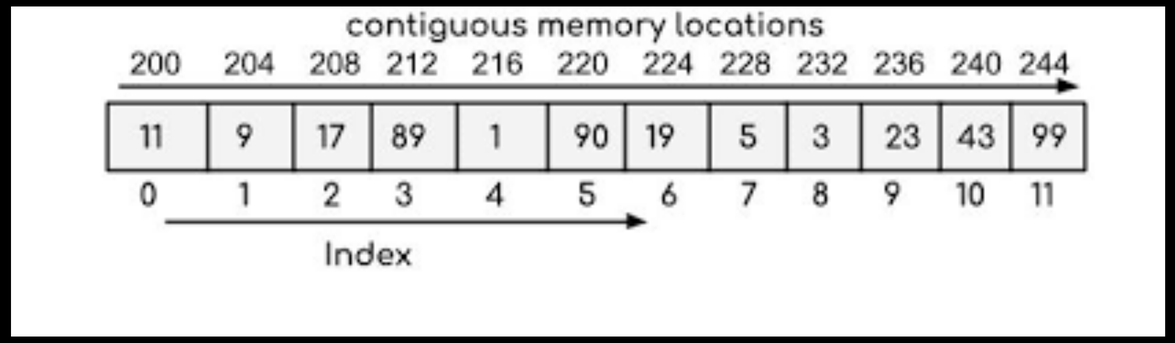
Conversion

- Conversion can be done both manually and automatically
- **autoboxing** - converting a primitive value into a corresponding wrapper object
- **unboxing** - converting a wrapper object into the corresponding primitive
- If we write a method that accepts a primitive value / wrapper object, we can still pass either value to it. Java will take care of the conversion and will be passing the right type (primitive or wrapper.

Arrays

Arrays

- Array: an ordered collection of primitives / objects of the same type
- each element has an index (starting from 0)
- Arrays have fixed size
- **java.util.Arrays** -> contains useful methods for array manipulation



String

String

- characters sequence
- internally backed by a **char-array**
- String are **immutable** – this means that once initialized, we cannot change it's value
- **StringBuilder** and **StringBuffer** (synchronized) can be used if we need mutable character sequences
- More on strings here

Static and final keywords

Static keyword

- Keyword: **static**
- Static structures belong to the class rather than the object
- Static:
  - **fields**: only one instance is created in memory; each instance can access and modify it
  - **methods**
  - **blocks**
  - **nested classes**
- When using a static field/method, you can use it without creating an instance

Final keyword

- Keyword: **final**
- Final:
  - **Attribute** / variable : cannot be modified once initialised
  - **Method**: cannot be overridden
  - **Class**: cannot be extended