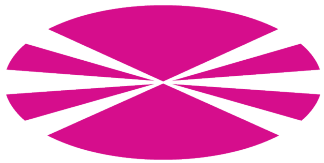
Practice 1: Introduction to Java

Software Design (614G01015)

Eduardo Mosqueira Rey (Coordinador)

Department of Computer Science and Information Technology
Faculty of Computer Science



UNIVERSIDADE DA CORUÑA

Table of Contents

Table of Contents

Origin

- **Oak programming language**. Developed at **Sun** by **James Gosling** and **Peter Naughton** to manage small electronic devices.
- Aspects sought: **hardware independence**, **"pure" and robust object-oriented** programming language, **based on C++**.
- Discontinued in Spring 1994 due to lack of profit.
- Relaunched in 1995 as Java, taking advantage of the Internet's hardware independence.
- In 2009, Oracle acquired Sun and it is the proprietary of the language.

James Gosling

Design goals of the Java PL

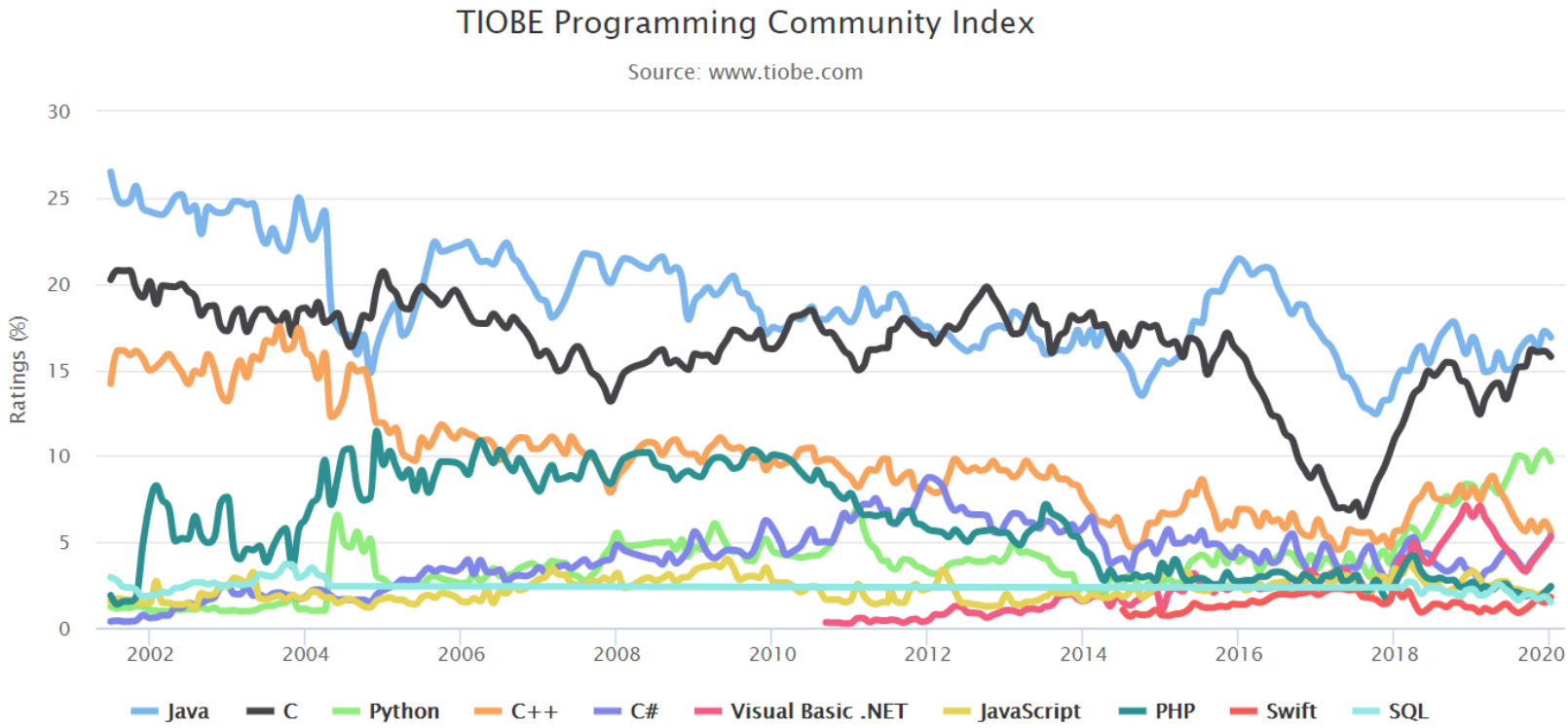
1. **Simple, object-oriented, and familiar**
   - Simple, easy to learn.
   - Pure object orientation.
   - Syntax is similar to C++. Familiar to developers.
2. **Robust and secure**
   - Multiple checks at compile time (e.g., non-initialized variables) and at runtime (e.g., out-of-bounds array indexes) to make programs more reliable and promote good programming practices.
   - Direct manipulation of pointers is not allowed, as it is a frequent source of errors.
   - Includes a *garbage collector* that frees unused memory.
   - Designed to work securely in distributed environments.

Design goals of the Java PL

3. **Architecture-neutral and portable**
   - Java applications can be run on multiple hardware/software architectures.
   - Java programs are compiled as intermediate *bytecodes*, which are later "interpreted' on each particular platform ("*write once, run everywhere*").
4. **High performance**
   - Despite being interpreted, Java performs optimizations in order to achieve high performance (*just-in-time* compilers, *HotSpot* technology, native code, etc.)
5. **Interpreted, threaded and dynamic**
   - Each platform has its corresponding *Java Virtual Machine (JVM)*, i.e., a virtual software processor that translates bytecodes to the machine code of the target architecture.
   - Java supports multi-threaded programming.
   - Even though Java is statically-typed, it supports object-oriented functionalities such as dynamic binding.

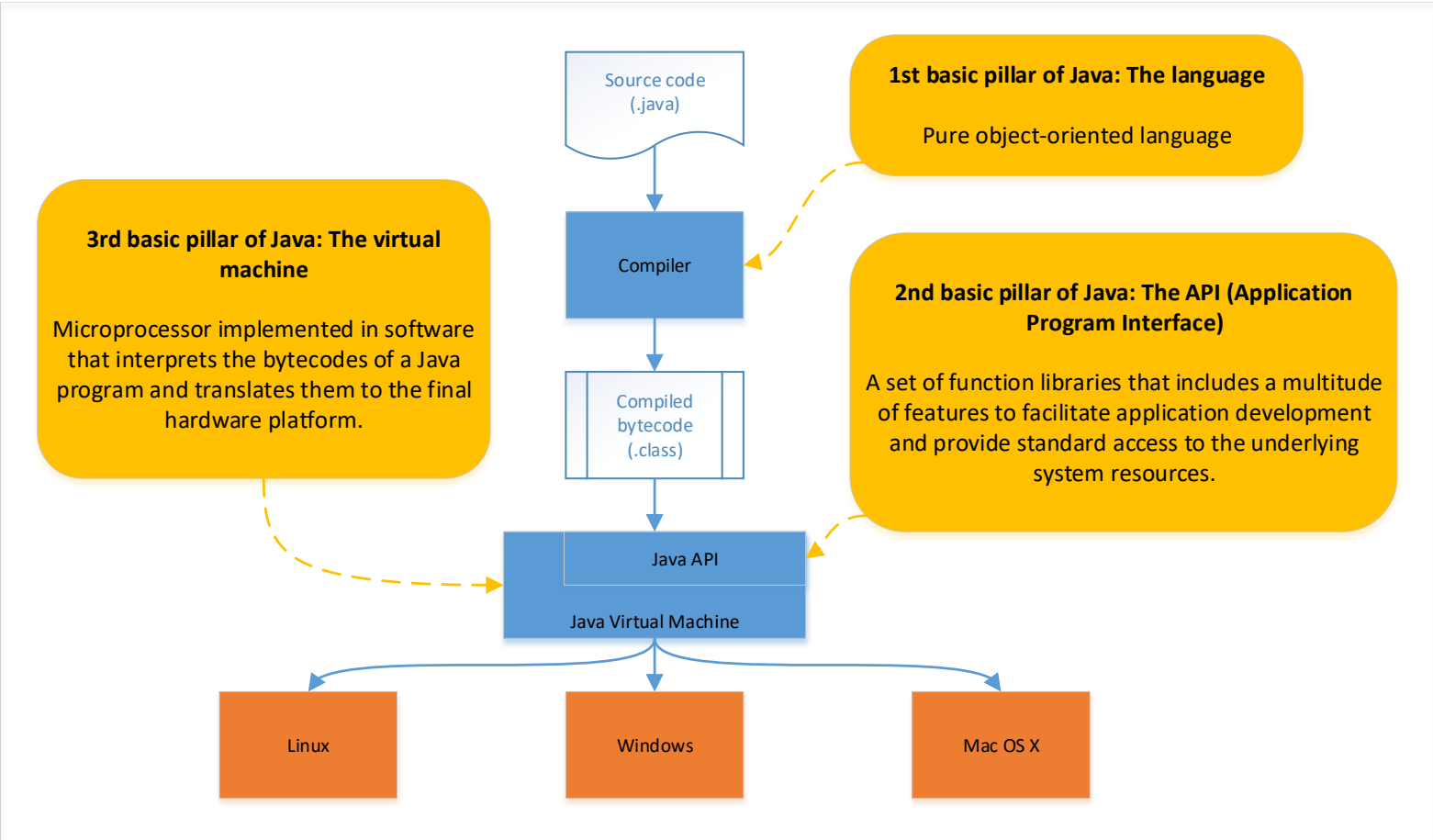
Java's popularity

- According to the TIOBE Index, Java is the most popular programming language from 2001 onwards (sometimes sharing the top spot with C).



Java Platform

- **The Java Platform**: Language, API and Virtual Machine.
- Distributed via a JDK (Java Development Kit).



Downloading the JDK

- **Oracle JDK**
  - Oracle distributes a version of the JDK under the *Oracle Technology Network (OTN)* license, which is only free for personal or non-profit use.
- **OpenJDK**
  - OpenJDK is the JDK version released under the GPL license.
  - Oracle JDK and OpenJDK are virtually equivalent. The only difference is the support offered by Oracle for its commercial version, which OpenJDK offers through other companies such as IBM.
  - **In this course we will use OpenJDK**. The most convenient source for the download is **AdoptOpenJDK:** **`https://adoptopenjdk.net/`**
  - We will also use **the latest version available**.

OpenJDK

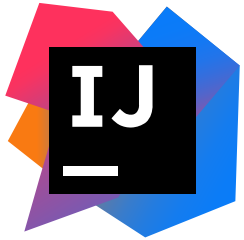
Other PLs with the platform

- The Java Language and the Java Virtual Machine are separate entities.
- You can use other PLs to write code, which will be later interpreted by the Java Virtual Machine:
  - **Kotlin**: A statically typed language and can be also compiled to JavaScript source code.
  - **Groovy**: A scripting language.
  - **Scala**: An OOPL that supports functional programming.
  - **Clojure**: a dialect of the Lisp functional programming language .



Table of Contents

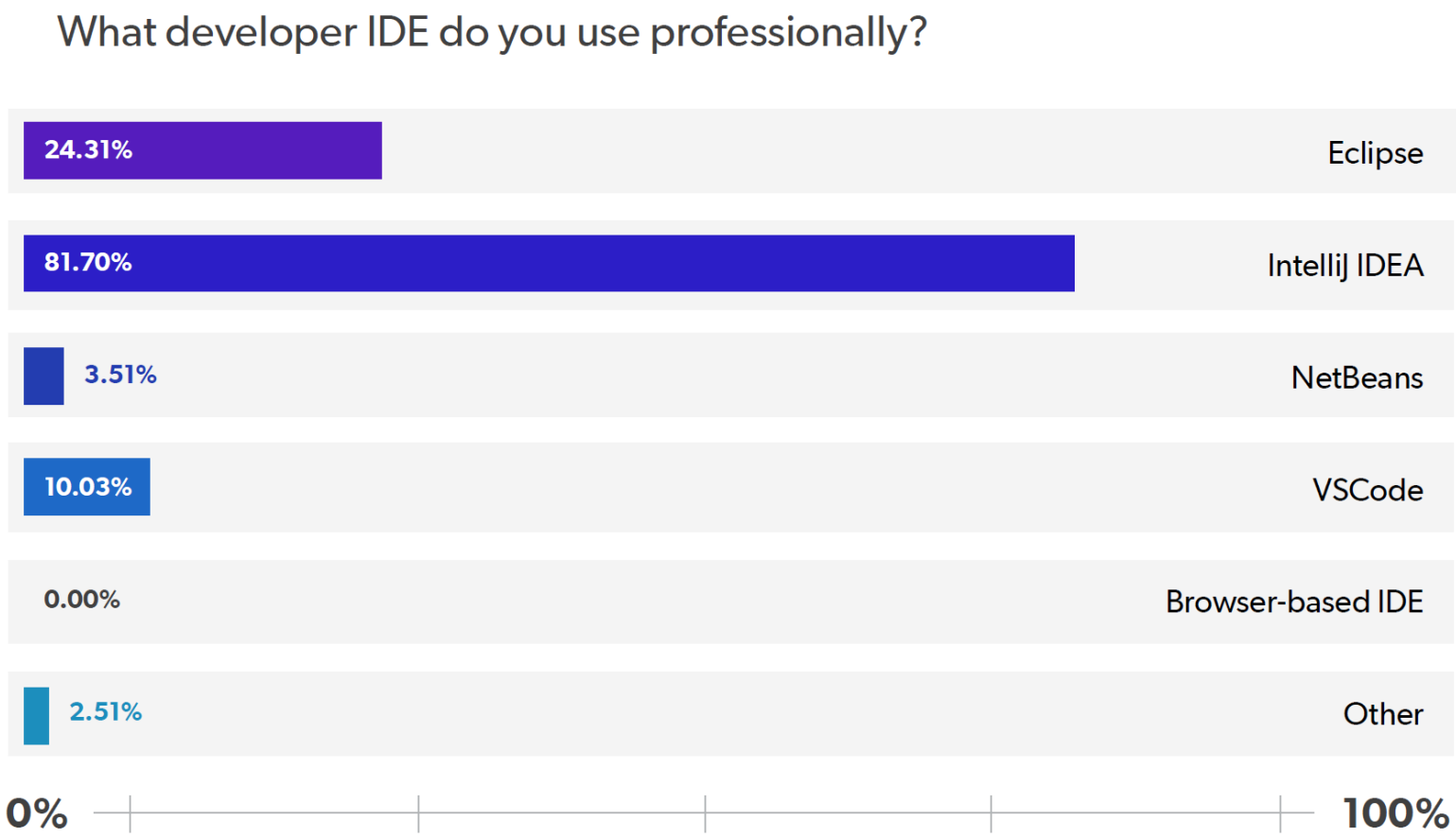
Introduction



- **IntelliJ IDEA** is an Integrated Development Environment (IDE) for Java (and other PLS for the JVM, such as Kotlin, Groovy or Scala). It was created by the JetBrains company.
- IntelliJ IDEA has **two editions**:
  - The ***Community*** edition is free and open-source (Apache 2.0 license) and is focused on basic projects for the JVM and Android development.
  - The ***Ultimate*** edition is non-free (although it can be used with an education license) and is focused on web and enterprise development.
- For this course, the ***latest Community edition*** will suffice. You can download it at:
  `https://www.jetbrains.com/es-es/idea/download/`

IntelliJ IDEA

- IntelliJ IDEA is currently the most widely used IDE for Java development[1].



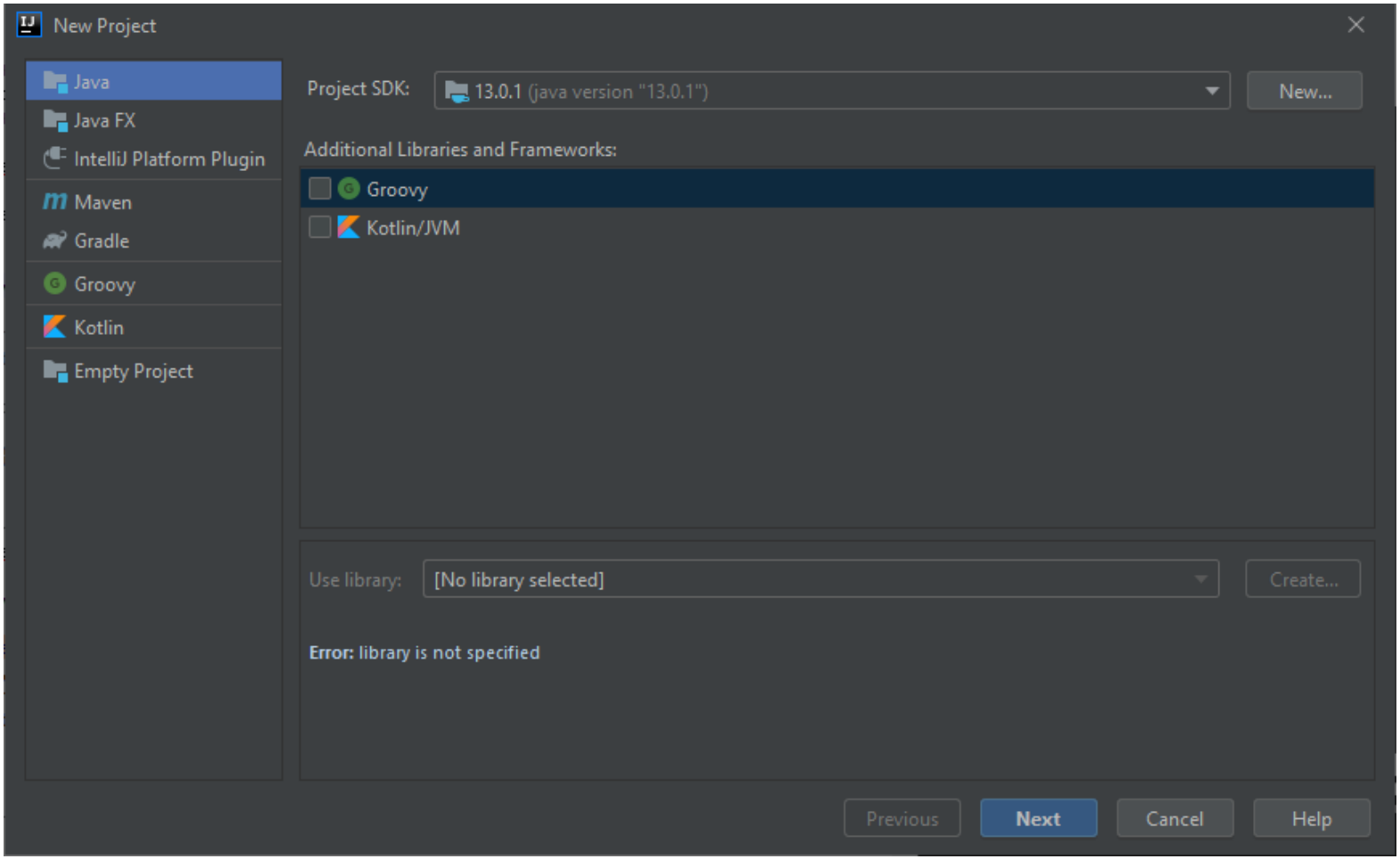
[1] Fuente: JRrebel Java Productivity Report 2020. https://www.jrebel.com/blog/2020-java-technology-report

IntelliJ IDEA

- The code is managed as projects. The first step is creating a new project.

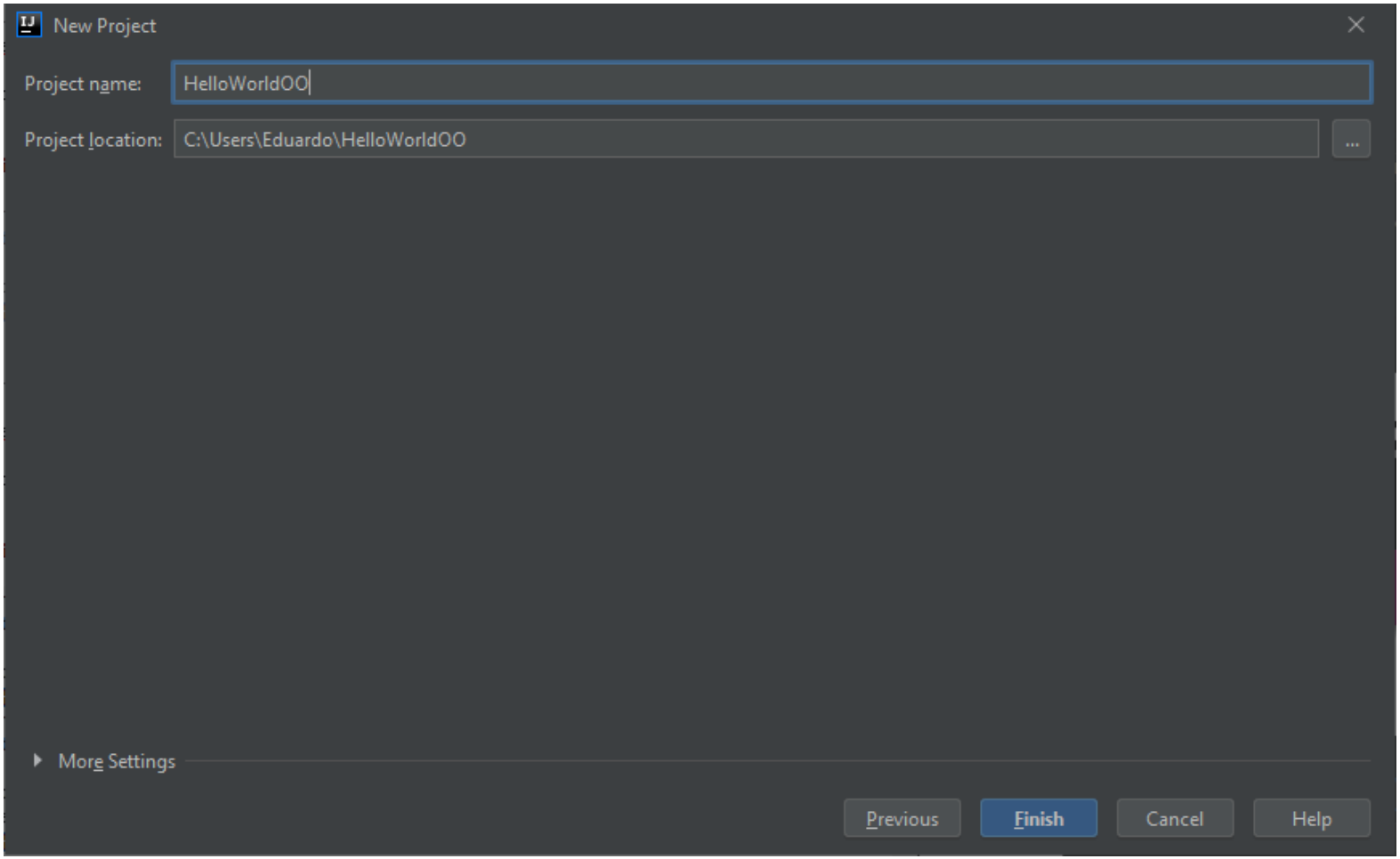
IntelliJ IDEA

- We choose creating a "Java" project, specifying the JDK.



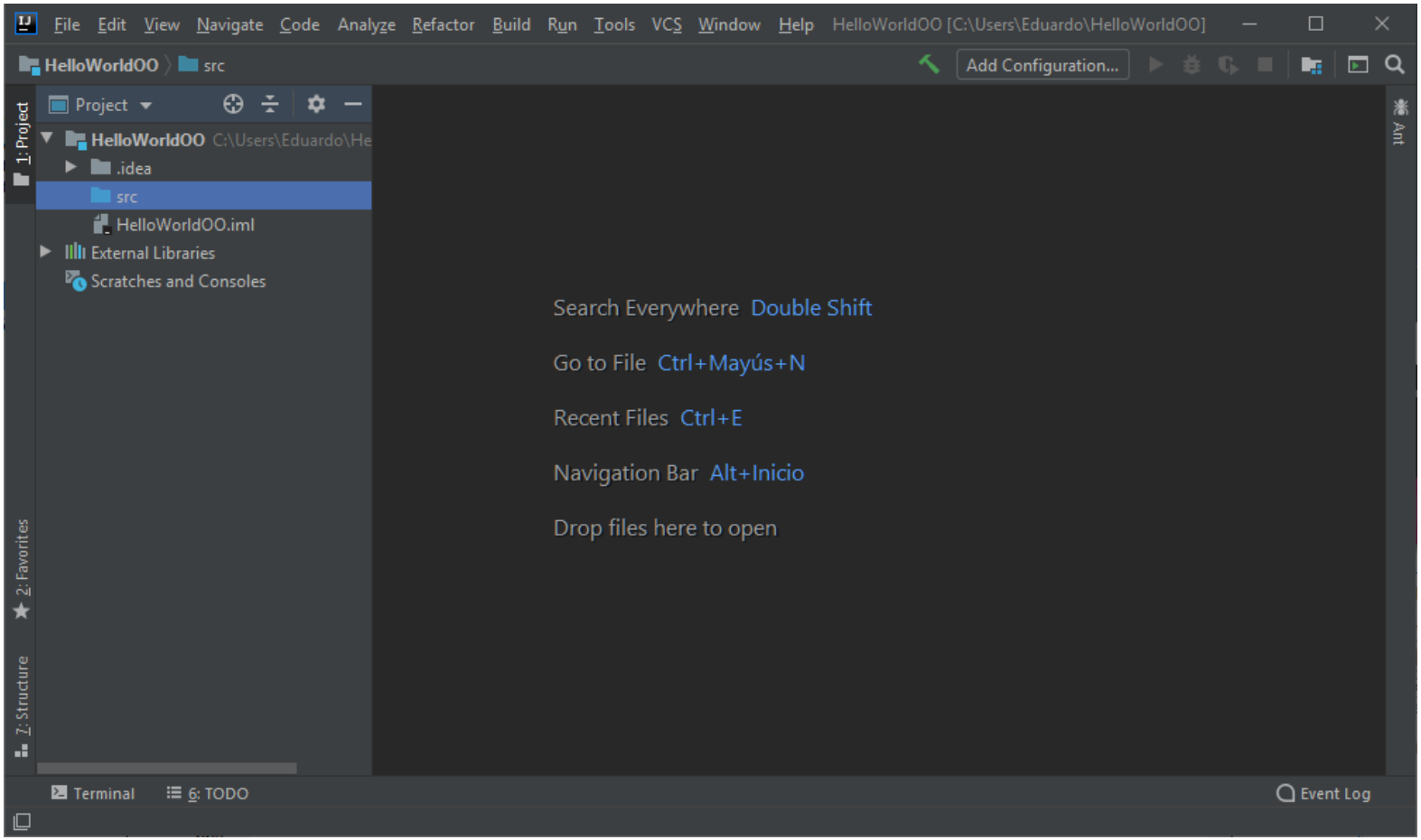
IntelliJ IDEA

- We enter the name and the location for the project.



IntelliJ IDEA

- The project is stored in the `.idea` folder, and the `.iml` file contains information about the *module* we are developing.
- The source files are stored in the `src` folder.



"Hello World" in Java

All Java code is inside a class

"Hello World" in Java

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

When we run a class, we run its `main` method

Prints "Hello World" on the terminal

"Hello, World" in Java

- **Our "Hello, World" code is a bad example of Object Orientation**:
  - A class is defined, but no object is instantiated.
  - Execution is delegated to a static `main` method. Static methods can be called without instantiating objects.
  - The class writes directly to the console.
- **An object-oriented code should include**:
  - Object instantiation.
  - Calls to non-static methods (i.e., methods that manipulate an object's state).
  - Reusable, architecture-independent classes that do not write directly to the console.

Object-Oriented "Hello World"

Object-Oriented "Hello World"

```
class HelloWorldOO {
    private String message = "Hello World";
    public String returnMessage() {
        return message;
    }

    public static void main(String[] args) {
        HelloWorldOO myHello = new HelloWorldOO();
        System.out.println(myHello.returnMessage());
    }
}
```

`getMessage` is not static (it needs an object in which it is executed) and returns a `String`

We create an object of the `HelloWorldOO` class using the `new` operator

`getMessage` retuns the message of the `myHello` object, the `println` method prints it in the console

Object-Oriented "Hello World"

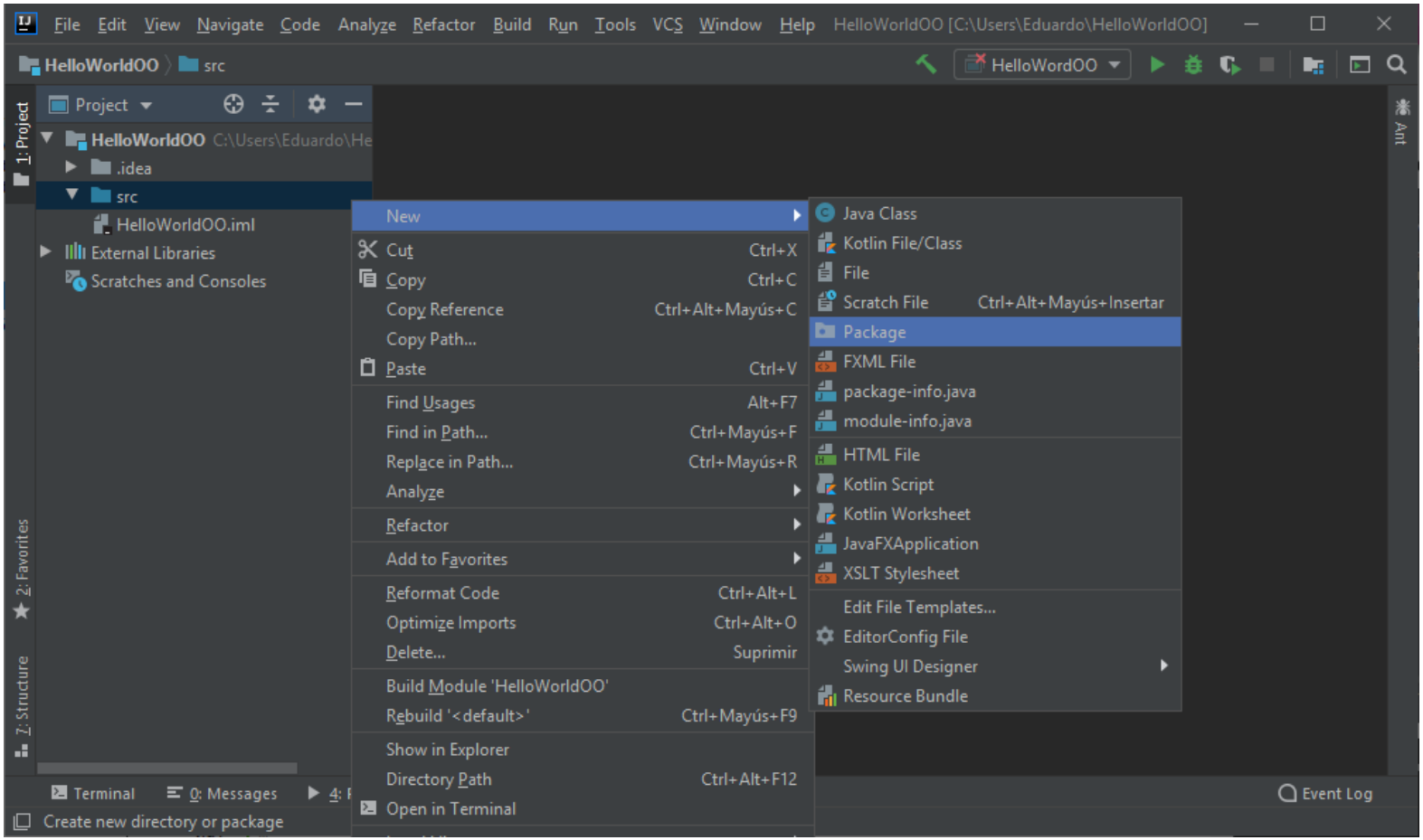
Object-Oriented "Hello World"

```
public class HelloWorldOO {
    private String message;

    public HelloWorldOO() {
        message = "Hello World";
    }

    public String returnMessage() {
        return message;
    }

    public static void main(String[] args) {
        HelloWorldOO myHello = new HelloWorldOO();
        System.out.println(myHello.returnMessage());
    }
}
```

Initialization can be done in constructor methods, that have the same name as the class and do not declare a return type

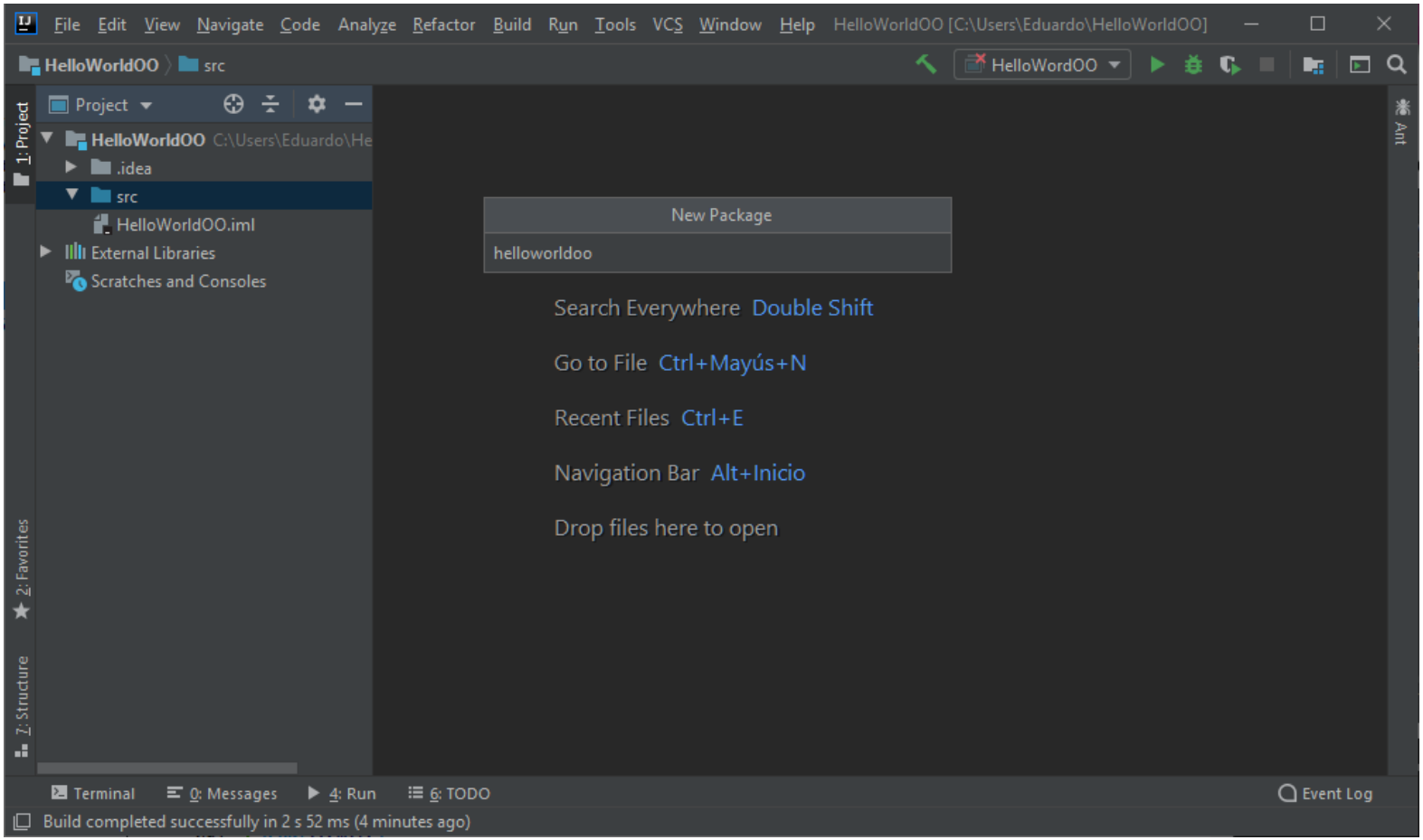
"Hello World" on IntelliJ IDEA

- It is recommended to arrange the source files into packages. Therefore, the first step will be creating one...



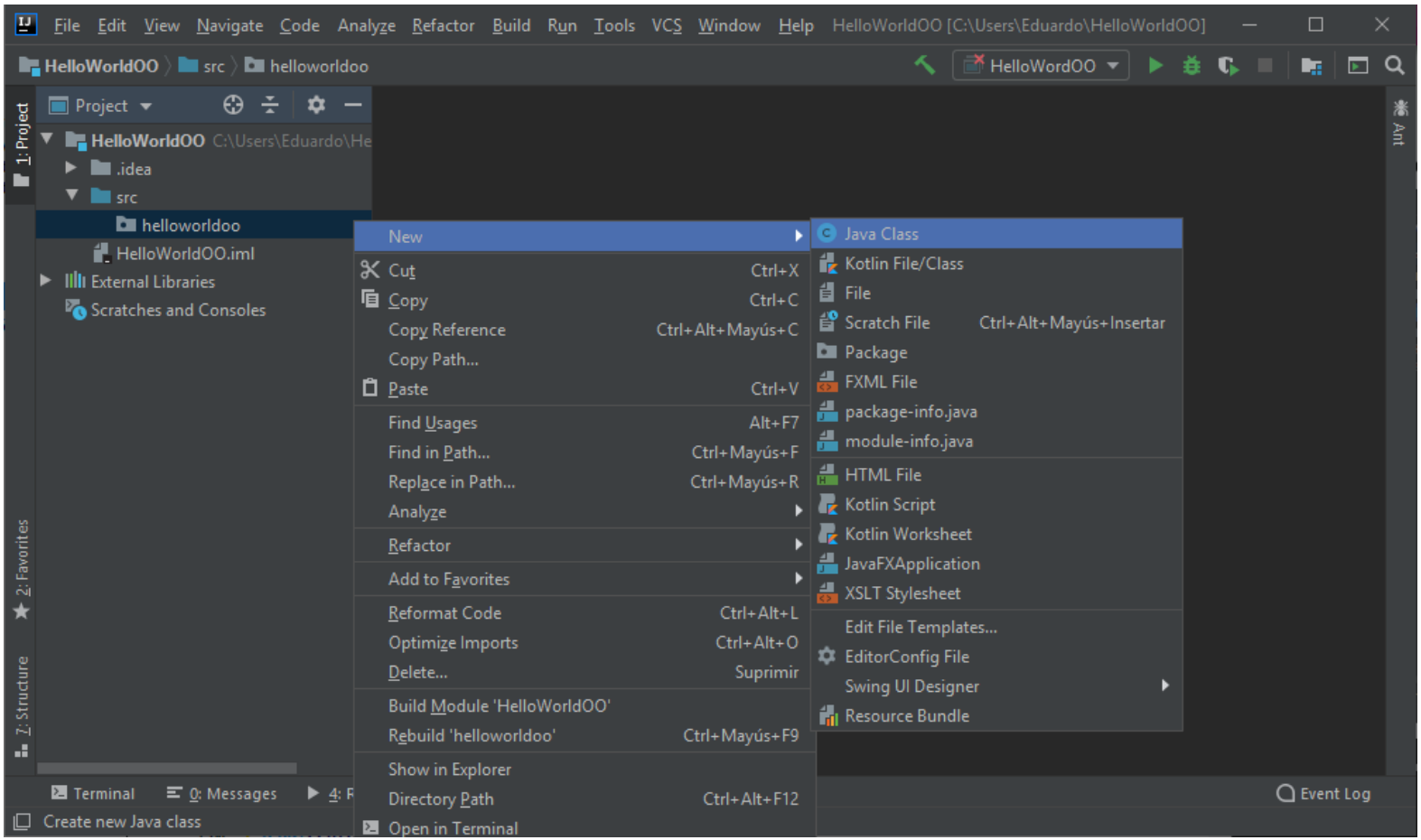
"Hello World" on IntelliJ IDEA

- ...which we will call `helloworldoo`.



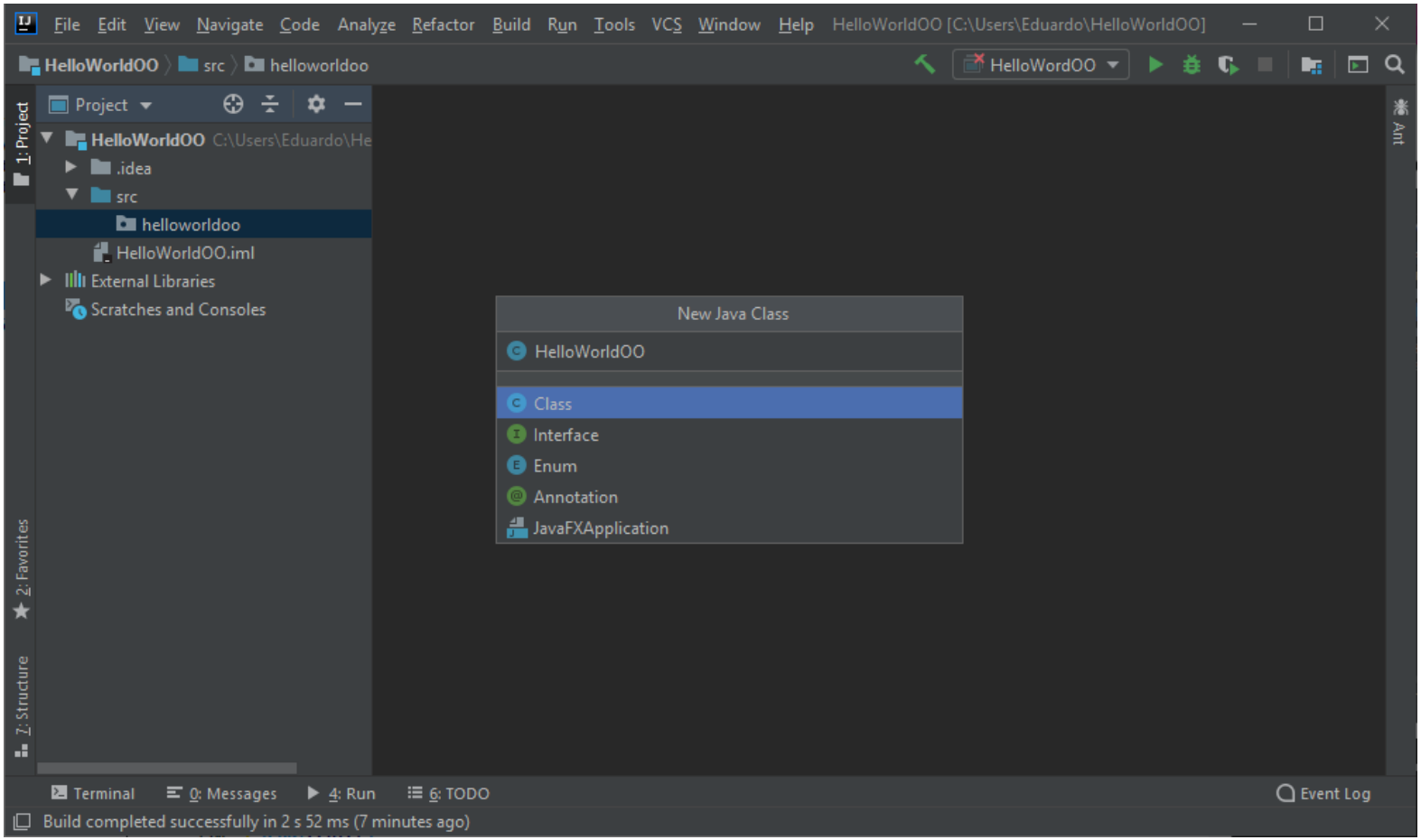
"Hello World" on IntelliJ IDEA

- Inside the `helloworldoo` package, we will create a new Java class...



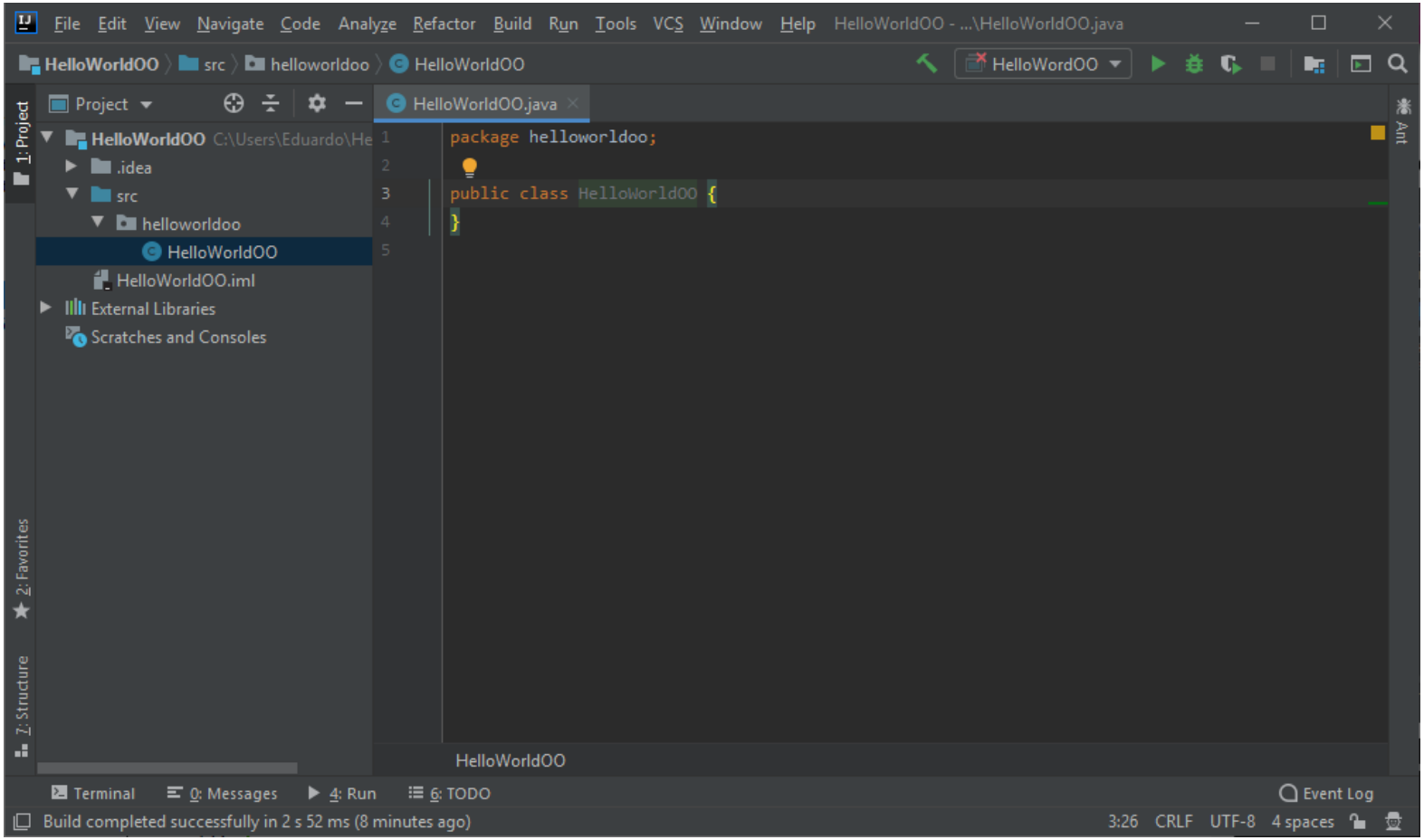
"Hello World" on IntelliJ IDEA

- ...which we will call `HelloWorldOO`.



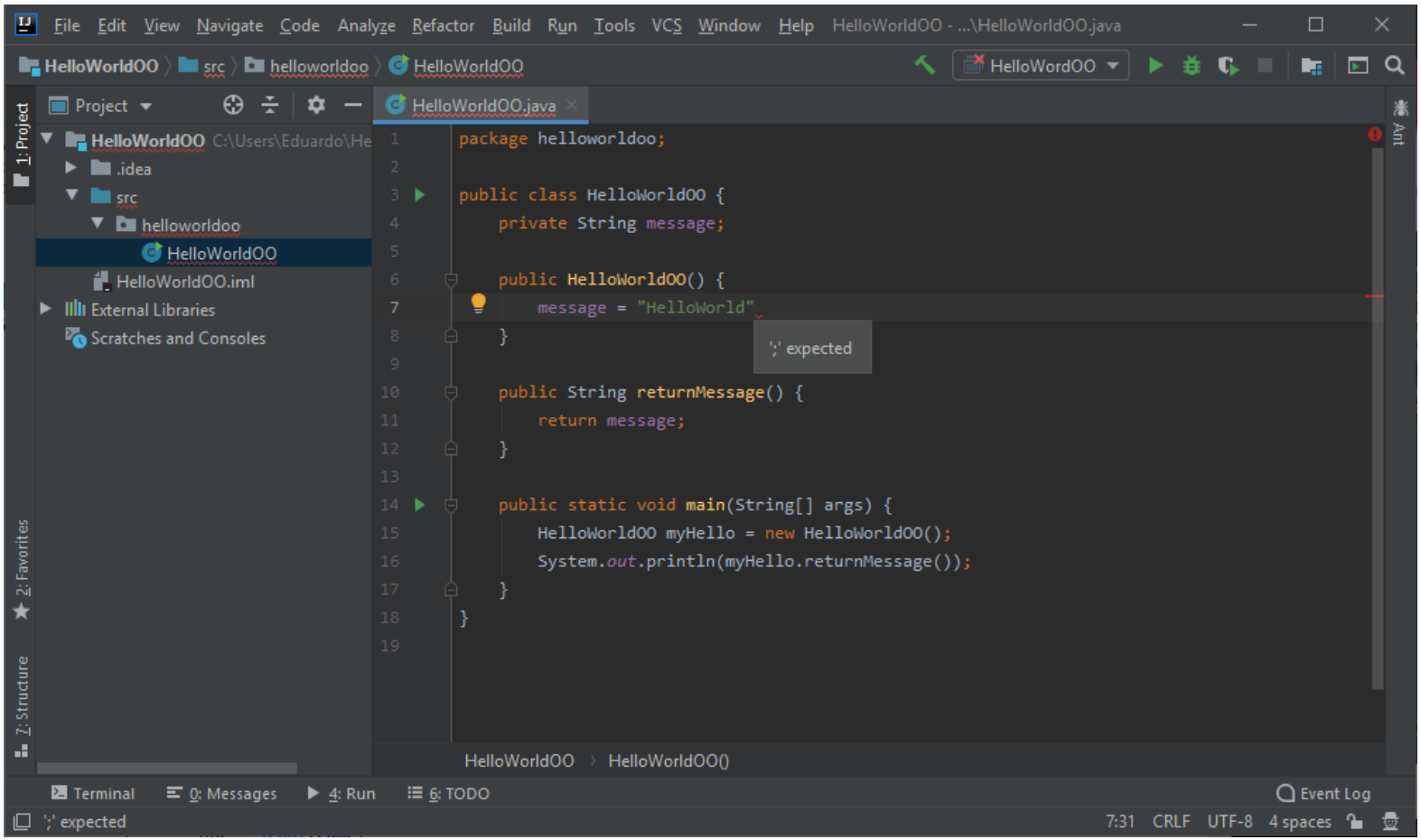
"Hello World" on IntelliJ IDEA

- The class is saved as the `HelloWorldOO.java` file.



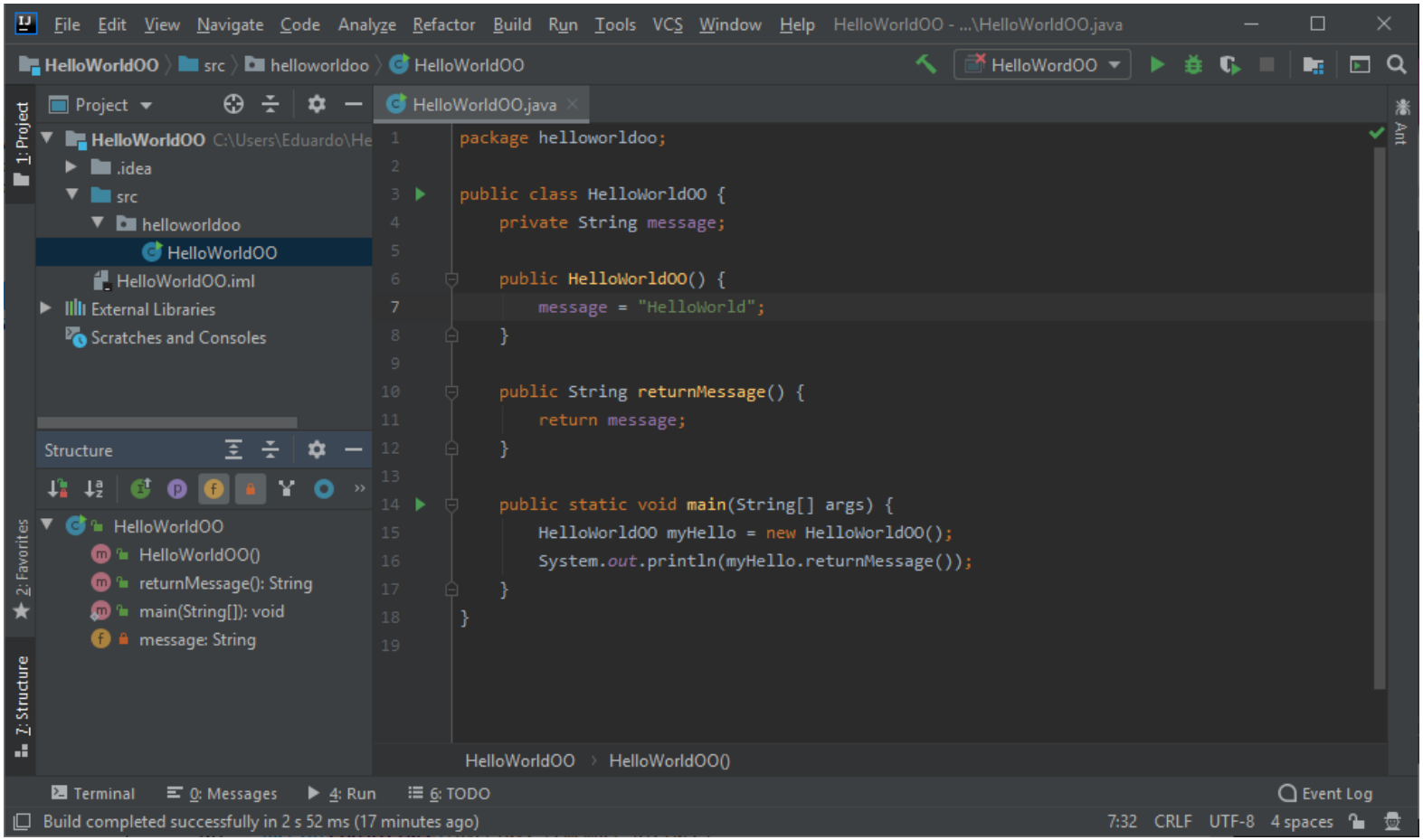
"Hello World" on IntelliJ IDEA

- The editor highlights in red the compilation errors as you write the code.



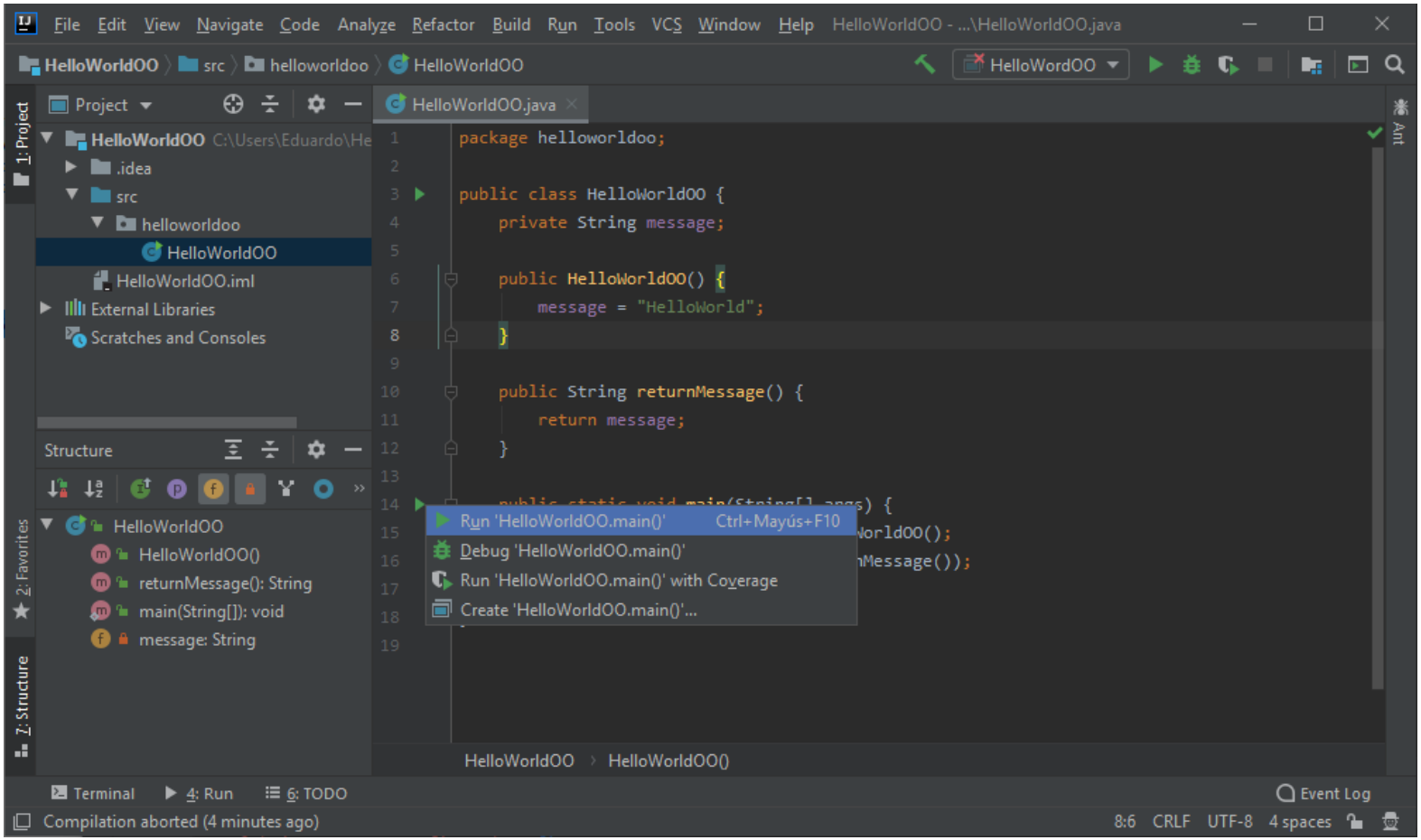
"Hello World" on IntelliJ IDEA

- The *Structure* window offers quick access to the contents of a class.



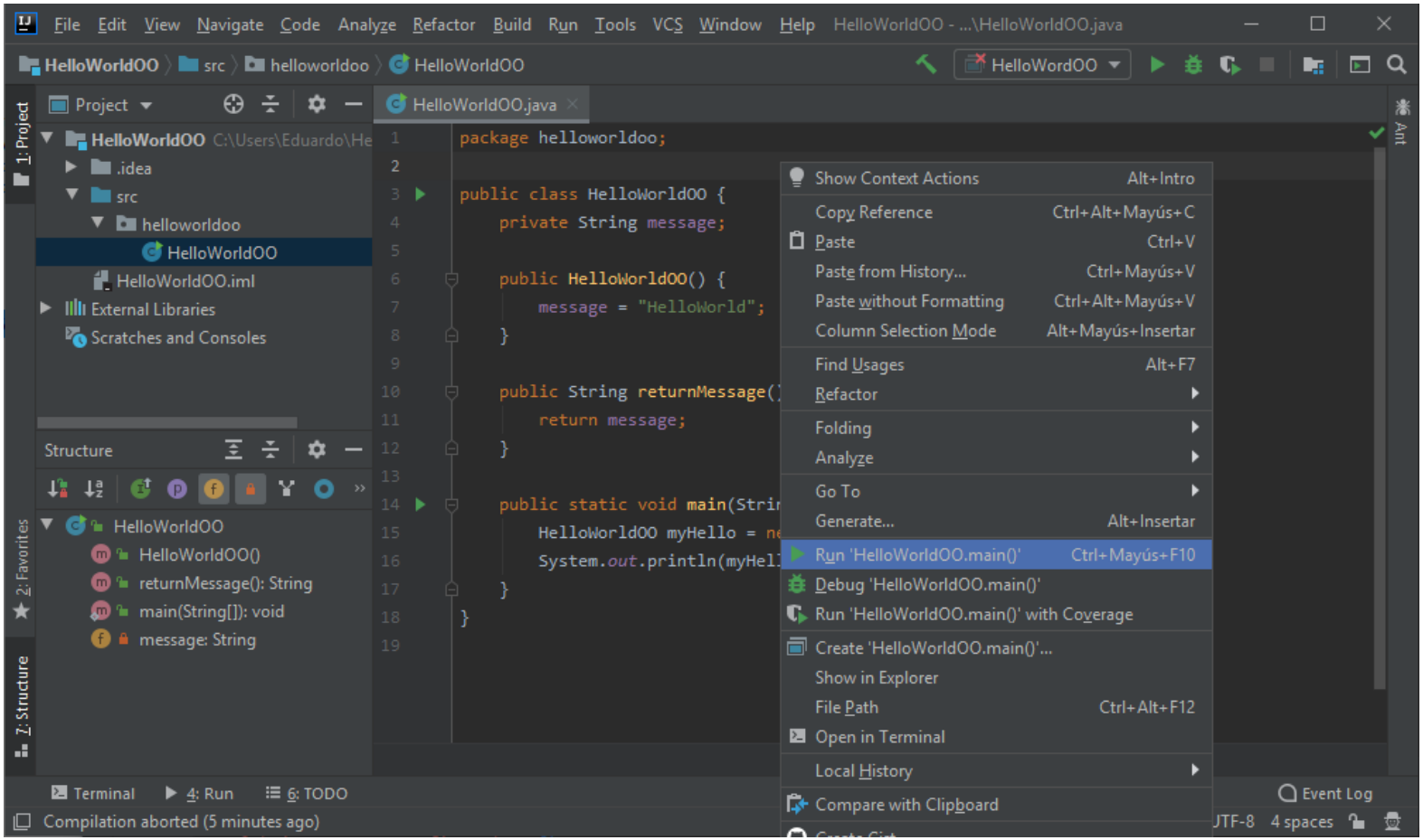
Compiling and running

- The code is being pre-compiled as we write it.
- To run a class with a `main` method, we can use the green triangle next to the `main`...



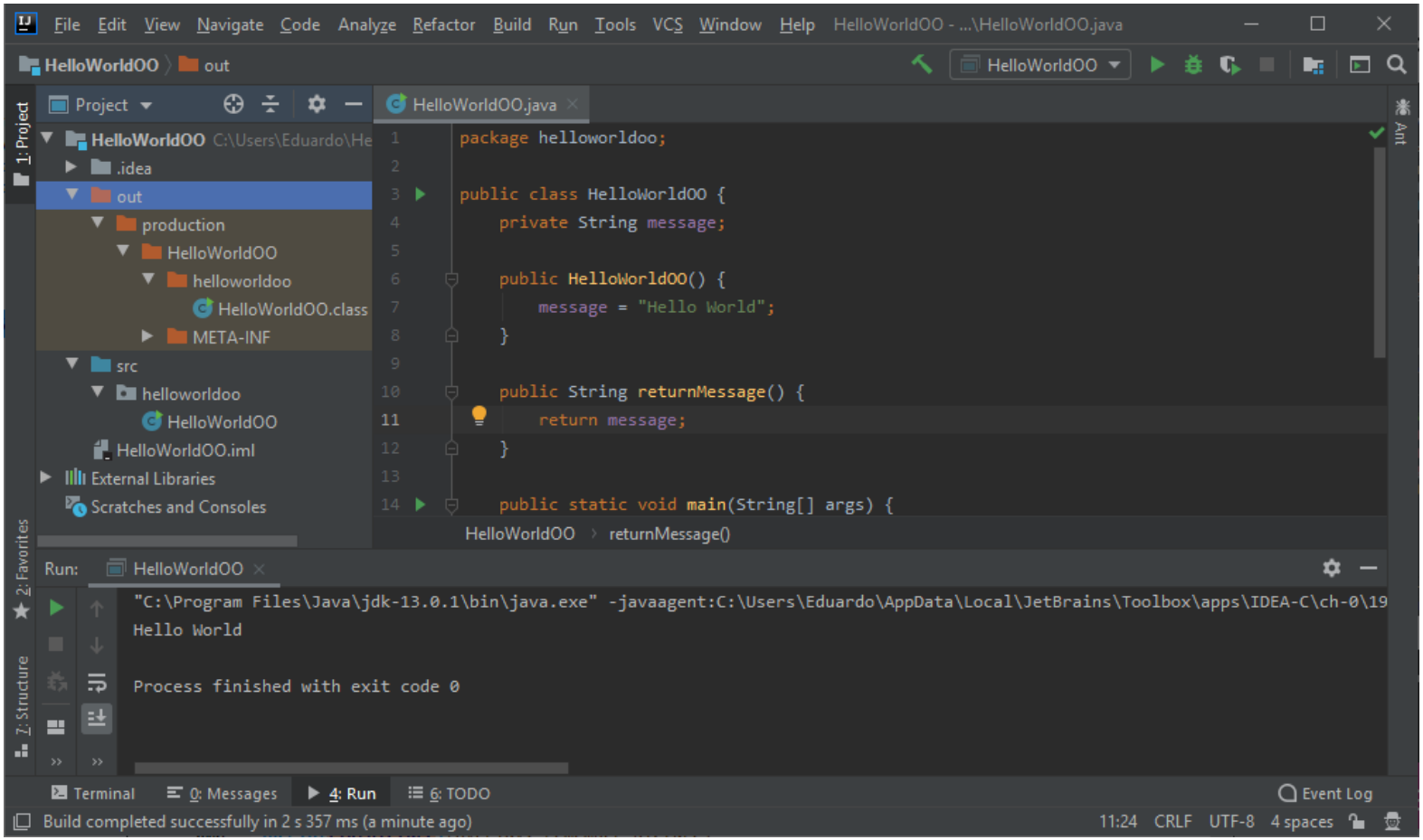
Compiling and running

- ... use the contextual menu when you right-click, or use the `Ctrl+Shift+F10` keyboard shortcut.



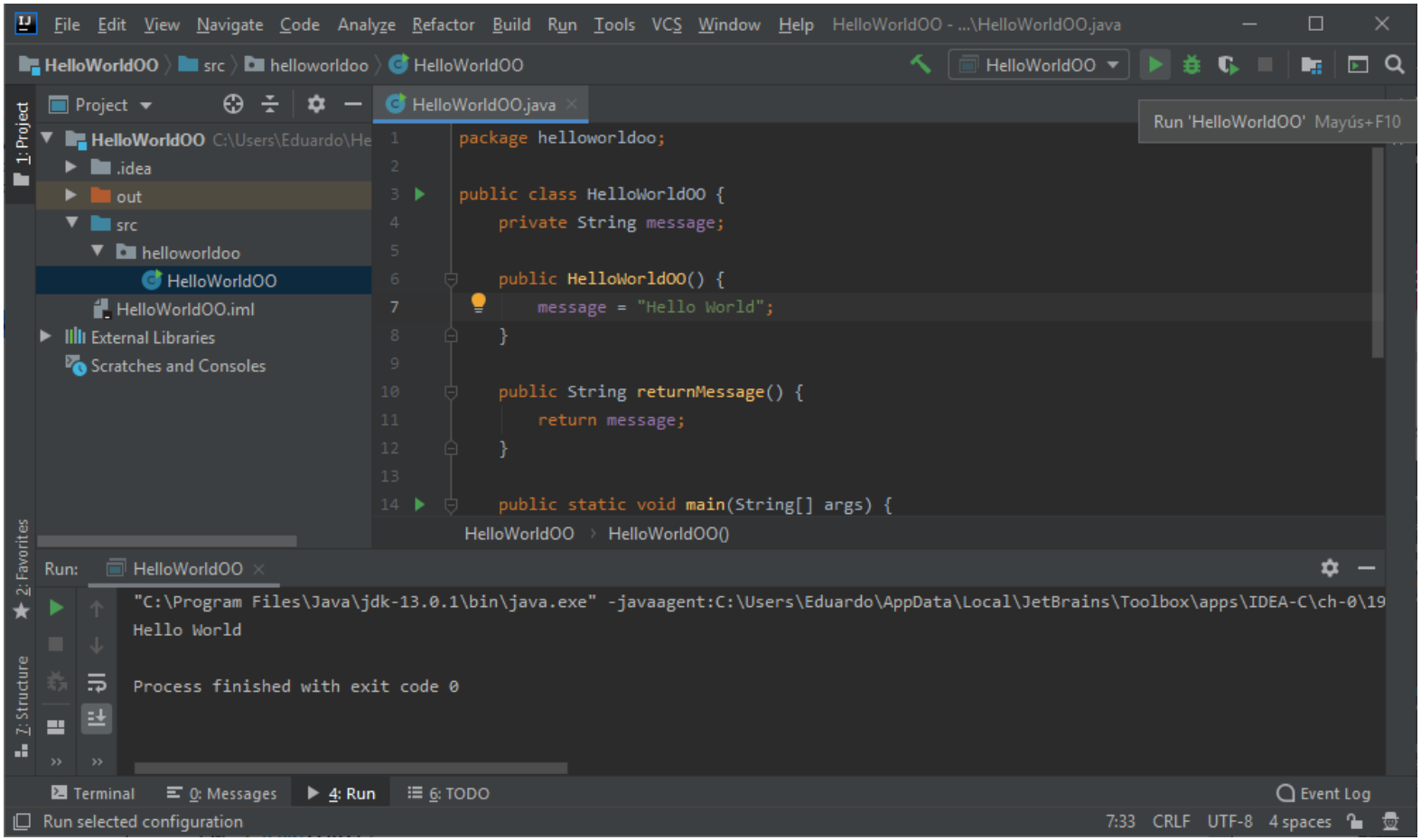
Compiling and running

- Running a project creates an `out` folder with the compiled `.class` files.
- The *Run* window is shown with the result of the execution.



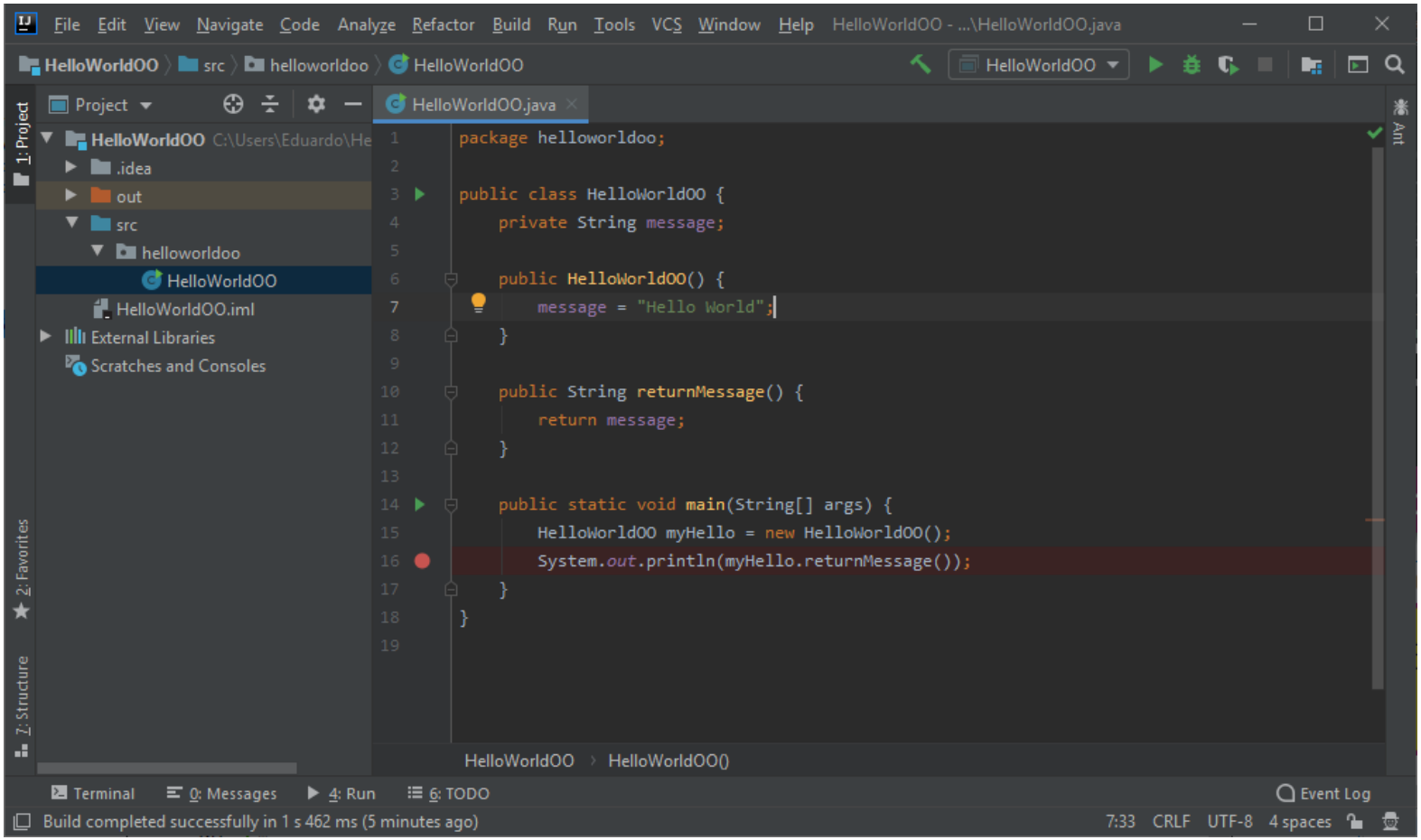
Compiling and running

- The first execution also creates a "*configuration*" that we can further customize or run again with `Mayús+F10` or the green triangle on the upper right corner.



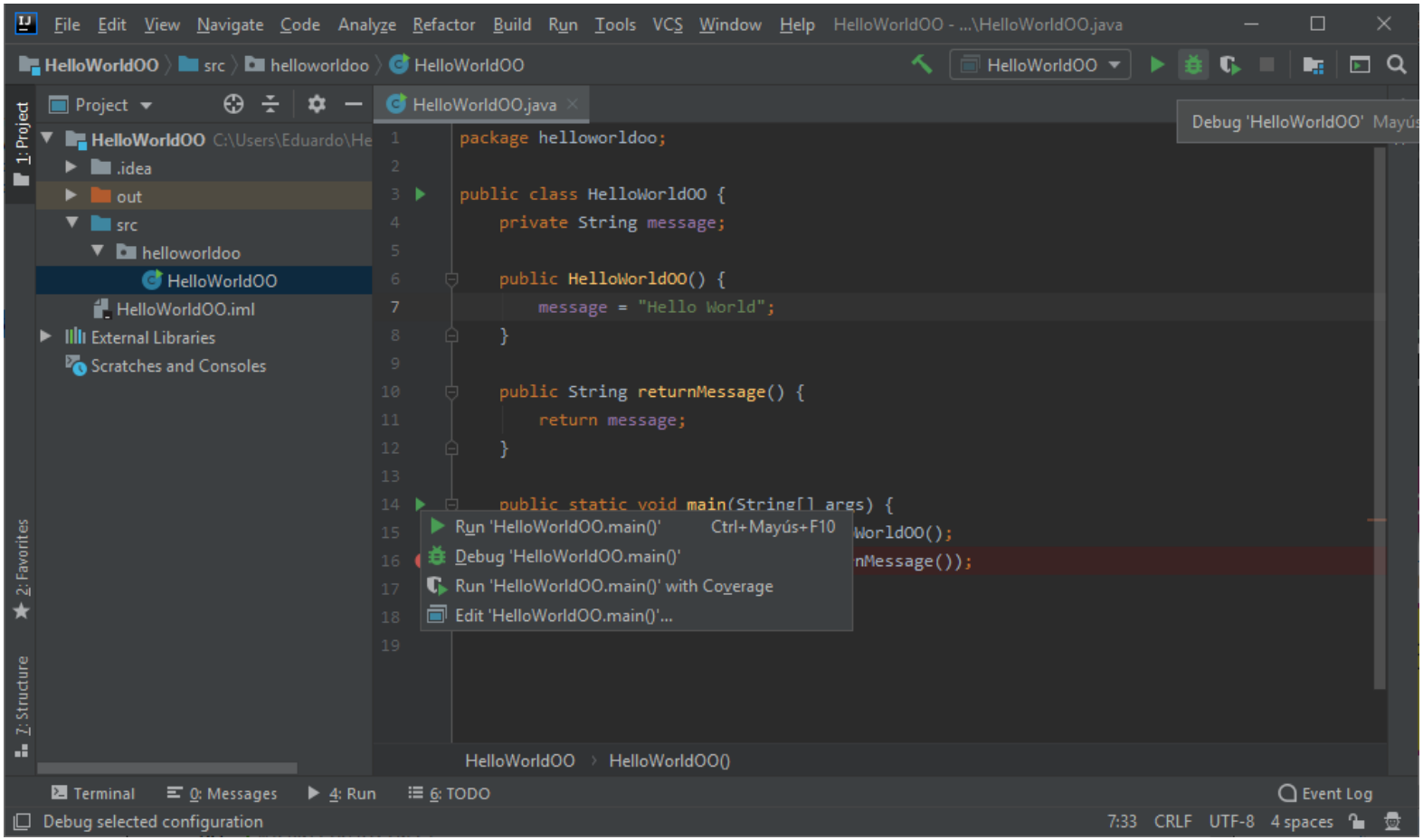
Debugging

- In order to (*debug*) code or execute it step by step, we must set a (*breakpoint*) by clicking next to the line number.



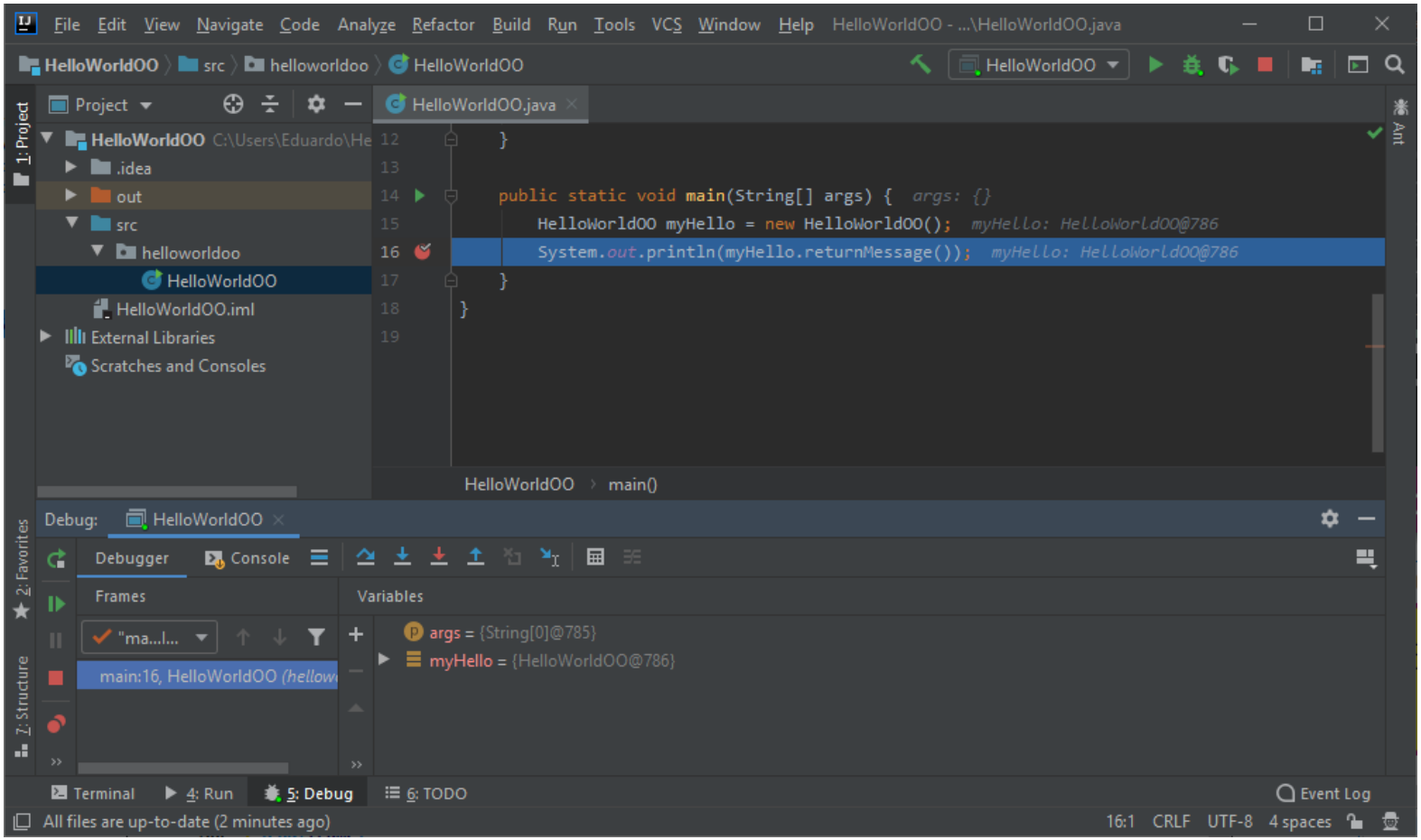
Debugging

- Clicking on the green *bug* icon, we start the debugging session.



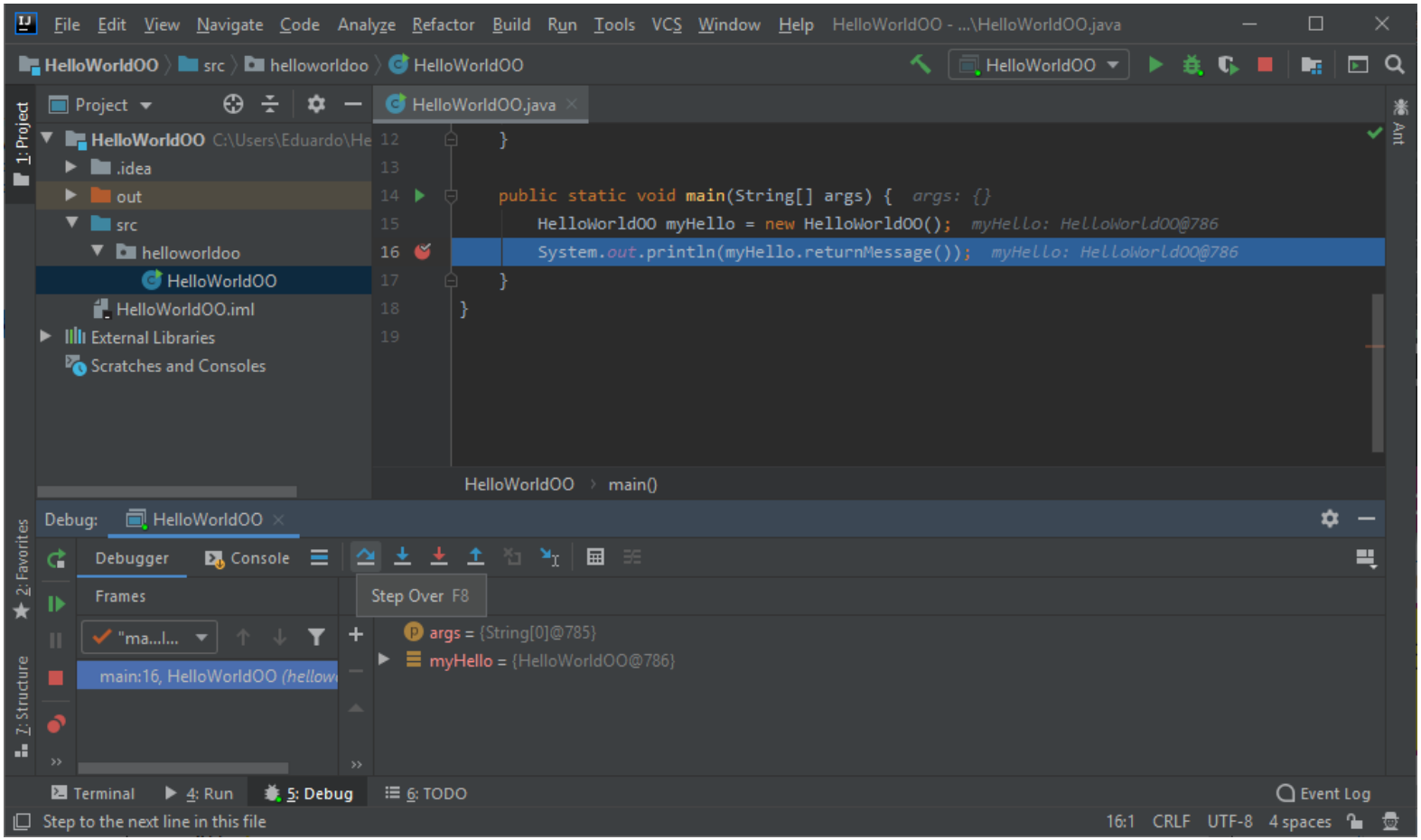
Debugging

- Execution stops at the *breakpoint*.
- A *Debug* window is shown, which lets us see the values of the variables and control the execution step by step.



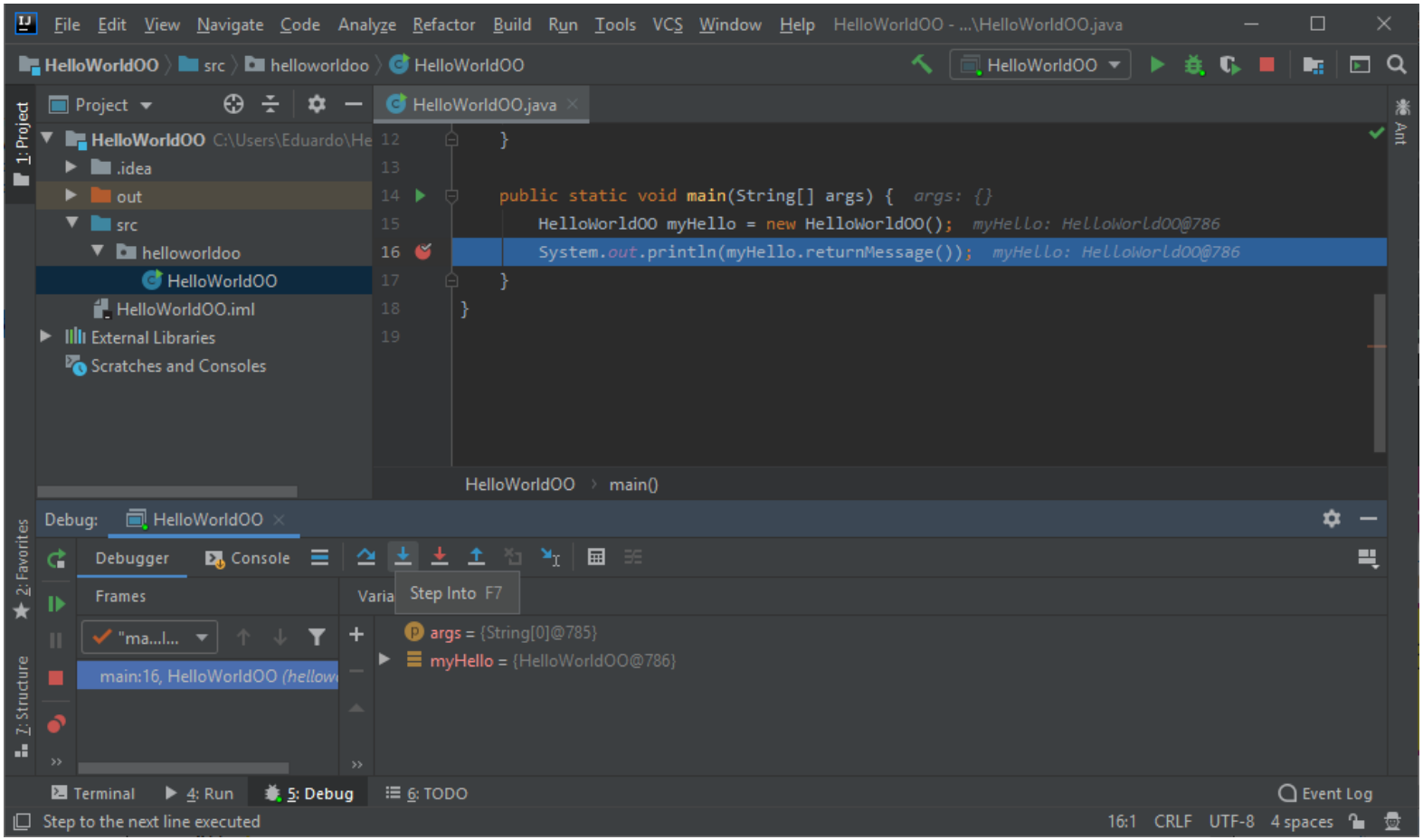
Debugging

- The execution is essentially controlled by the `F8` key or *Step Over*, which runs a method directly (not step by step)...



Debugging

- ... and using the `F7` key or *Step Into*, which executes a method step by step.



Debugging

- Debugging ends when the execution ends, or at any point if we click on the *Stop* button (`Ctrl+F2`).

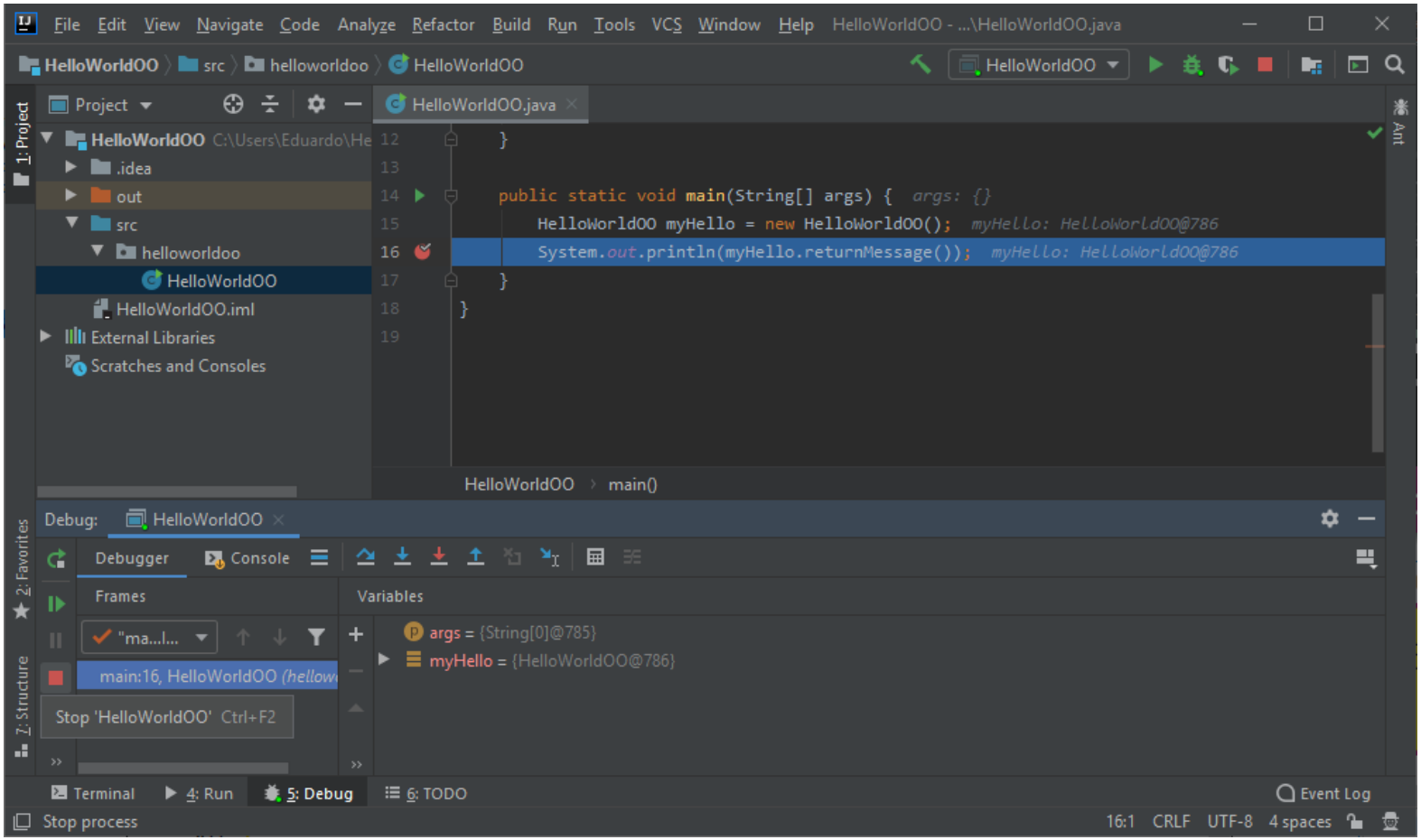


Table of Contents

Naming conventions

- Java is case-sensitive.
- Identifiers typically use the **CamelCase** notation (i.e., no spaces between letters, mixes uppercase and lowercase).
- **Following these conventions is important (failing to do this may be penalized in your grade.)**.
- First letter is in uppercase for classes (e.g., `GeometricFigure`), and in lowercase for variables (e.g., `localVariable`).

Naming conventions

Indentifier	**Rule**	**Example**
Classes and interfaces	Nouns; CamelCase starting with a capital letter	`class ImageSprite`
Methods	Verbs; CamelCase starting with a lowercase letter	`runFast();` `getBackground();`
Variables	Nouns; CamelCase starting with a lowercase letter	`int widthFigure;` `int heightFigure;`
Constants	All caps (no CamelCase); underscores for spacing	`int MIN_WIDTH=1;` `int MAX_WIDTH=99`
Packages	Lowercase (no CamelCase); short names	`java.awt.datatransfer,` `java.util`

Primitive data types

Type	Size	Range	Format	Example
byte	8 bits	-128..127	Two's complement	`88`
short	16 bits	-32.768..32767	Two's complement	`-1578`
int	32 bits	$-2^{31}..2^{31}-1$	Two's complement	`423424`
long	64 bits	$-2^{63}..2^{63}-1$	Two's complement	`123456L`
float	32 bits	..[2]	IEEE 754	`126.5f`
double	64 bits	..	IEEE 754	`126.5d`
char	16 bits	0..65.535	Unicode	`'a'`
boolean	1 bit[3]	[true/false]	Binario	`false`

[2] Consult `https://docs.oracle.com/javase/specs/jls/se14/html/jls-4.html#jls-4.2.3`

[3] Real size not stated

Arrays

- Arrays are similar to those of C, but with significant differences.
- It is possible to declare an array and assign values immediately (as in C).
- But it is also possible to declare an array of unspecified size and specifying it later using the `new` operator.

Arrays

```
int[] numbers1 = {1, 4, 5, 2, 7}; // as in C

int[] numbers2; // not size declared
numbers2 = {8, 7, 2 ,9, 4}; // ERROR: array literals only in declaration
numbers2 = new int[5]; // allocating 5 positions with "0" values
System.out.println(numbers2[2]); // Prints 0
```

Arrays

- Arrays are pointers, so we can dinamically change the address they point to.
- We can use it to make them "grow" copying their data to bigger arrays (costly but useful).
- Accessing an index outside of the dimensions of the *array* causes a runtime error.

Arrays that *grow*

```
int[] numbers1 = {4, 9, 3, 6, 2}; // five positions
int[] numbers2 = new int[10];
for (int i=0; i<numbers1.length; i++) {
    numbers2[i] = numbers1[i]; // copy items
}
numbers1 = numbers2; // numbers1 points to numbers2
System.out.println(numbers1[0]); // 4
System.out.println(numbers1.length); // numbers1 has 10 positions

numbers1[15] = 42; // Throws ArrayIndexOutOfBoundsException
```

Matrices

- A matrix is actually an array of arrays.

Matrices

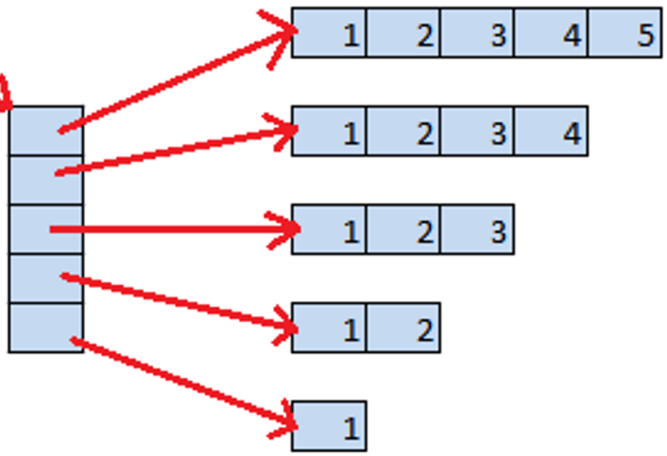
```
int[][] matrix = {{1, 2, 3},
                  {4, 5, 6},
                  {7, 8, 9}};
System.out.println(matrix[1][2]); // 6
```

	Column 0	Column 1	Column 2
Row 0	matrix[0][0] **1**	matrix[0][1] **2**	matrix[0][2] **3**
Row 1	matrix[1][0] **4**	matrix[1][1] **5**	matrix[1][2] **6**
Row 2	matrix[2][0] **7**	matrix[2][1] **8**	matrix[2][2] **9**

Matrices

- Unlike C, the rows do not necessarily have the same length.
- This results in *ragged arrays*.

```
int[][] triangleArray = {
        {1, 2, 3, 4, 5},
        {2, 3, 4, 5},
        {3, 4, 5},
        {4, 5},
        {5},
};
```



Strings

- Represents a sequence of characters.
- Can be easily instantiated using *literals*:
  `String s = "Hello World";`
- Immutable (once created, they cannot be modified).
- There are functions for obtaining a new String that is a modified version of another String. e.g. `toUpperCase()`.
- The mutable (modifiable) version of a String is represented by the class `StringBuilder`.

`String` and `StringBuilder`

```
String s1 = "Hello World"; // String literals
String s2 = s1.toUpperCase(); // s1 is immutable
System.out.println("s1 = " + s1); // s1 = Hello World
System.out.println("s2 = " + s2); // s2 = HELLO WORLD

StringBuilder sb = new StringBuilder("Hello");
sb.append(" World"); // StringBuilder is mutable
System.out.println("sb = " + sb); // "sb = HelloWorld";
```

Strings

- Special characters are added to `Strings` using *escape sequences*: tabulation (`\t`), new line (`\n`), double quotes (`\"`), single quotes (`\'`), backslash (`\\`), etc.
- These escape sequences can be avoided by using *text blocks* (Java 15) defined with triple quotation marks (whitespace to the left is ignored).

Escape sequences and text blocks

```
String query1 =
        "SELECT name, age\n" +
        "FROM EMP\n" +
        "WHERE name = \'John\'\n" +
        "\tAND age > 20";

String query2 = """
                SELECT name, age
                FROM EMP
                WHERE name = 'John'
                    AND age > 20
                """;
```

Operators

Type	**Operator**	**Description**
Binary arithmetic	+, -, *, /, %	
Unary arithmetic	++, - -	Increment, Decrement
Equality	==, !=	Equal, Not equal
Relational	>, >=, <, <=	
Conditional	&&, ||	Logical AND, Logical OR
Type comparison	instanceof	Whether an object is an instance of the class
Bitwise operation	&, |, ^, ~	AND, OR, XOR, NOT
Bit shifting	>>, <<, >>>	Right, Left, Unsigned right
Assignment	=, (operator)=	Basic, Cumulative

Control Structures

- The syntax for control structures is basically identical to that of C, with some later additions.
- If the block has only one instructions, curly brackets are not necessary (but they're recommended to avoid ambiguity and errors).

Conditional structures `if..(then)..else`

```
// if..(then)
if (isMoving) {
    currentSpeed++;
}

// if..(then)..else
if (isMoving) {
    currentSpeed++;
} else {
    currentSpeed = 0;
}
```

Control Structures

- The switch statement is similar in C and Java, but it's too error-prone (it's easy to forget the `break`) .

Traditional `switch` statement

```
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        numLetters = 6;
        break;
    case TUESDAY:
        numLetters = 7;
        break;
    case THURSDAY:
    case SATURDAY:
        numLetters = 8;
        break;
    case WEDNESDAY:
        numLetters = 9;
        break;
    default:
        throw new IllegalStateException("Wat: " + day);
}
```

Control Structures

- Since Java 14, you can use `switch` statements that let you assign the result of a `switch` to a variable.
- They avoid the controversial `break`.
- They check at compile time if all possibilities are covered, making the `default` clause unnecessary.

Conditional structure with `switch` expressions

```
numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
    case WEDNESDAY              -> 9;
};
```

Control Structures

- `While` loops are also similar to those of C.

`While` loops

```
// while loop
int count = 0;
while (count < 10) {
    count++;
}
System.out.println("count = " + count);

// do..while loop
count = 0;
do {
    count++;
} while (count < 10) ;
System.out.println("count = " + count);
```

Control Structures

- Java has two types of `for` statements. One is analogous to the C `for`.
- The other is called the `for-each` loop and is used to iterate through collections of objects ⇒ **Unit 3**.

`For` loops

```
// Classic for
for(int i = 0; i<10; i++) {
    System.out.println(i);
}

// For-each loop
int[] data = {0, 1, 2, 3, 4, 5};
for(int i : data) {
    System.out.println(i); // 0 1 2 3 4 5
}
```

Control Structures

- `break` and `continue` are also like in C.

For loops

```
System.out.println("break and continue");
for(int i = 0; i<10; i++) {
    if (i == 5)
        continue;
    if (i == 8)
        break;
    System.out.println(i); // 0 1 2 3 4 6 7
}
```

Control Structures

- Java also lets you use `break` and `continue` to jump to labeled locations.
- They're a controlled *goto* that jumps to a specified, non-arbitrary place when using nested loops.

For loops

```
int[][] arrayOfInts = {{ 32, 87, 3}, {12, 1076, 2000}, { 622, 127, 77}};
int i = -1;
int j = -1;
int searchFor = 12;
SEARCH: // LABEL
for (i = 0; i < arrayOfInts.length; i++) {
    for (j = 0; j < arrayOfInts[i].length; j++) {
        if (arrayOfInts[i][j] == searchFor) {
            break SEARCH; // Break the two loops
        }
    }
}
System.out.println("Found " + searchFor + " at [" + i + ", " + j + "]");
```

Comments

Comments

```
public class Class {
    /* Variables */
    private int x;
    private int y;
    /*
     * Methods
     * (reading and writing)
     */
    public int getX() { return x; }

    /**
     * Assigns the value i to the attribu
     * (if greater than zero)
     * @param i The value that will be assigned to x
     */
    public void setX(int i) {
        if (i < 0) {
            x = 0; // Negative values are not allowed
        } else {
            x = i; // i is assigned to x
        }
    }
}
```

Block comment, one line

Block comment, multiple lines

Javadoc comment

Line comment

Javadoc comments

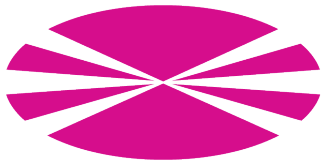
- Javadoc comments start with `/**`
- Placed above the corresponding class or method.
- May include standardized tags (e.g., `@param`, `@return`, `@author`, etc.)
- The `javadoc` tool reads these comments and tags and generates HTML-formatted documentation (e.g., the Java API Specification at `https://docs.oracle.com/en/java/javase/11/docs/api/index.html`).
- Integrated Development Environments (IDEs) simplify the task of writing Javadoc comments.
- **Javadoc comments are not mandatory in this course, but we expect regular comments in the code, especially to explain complicated fragments..**

Practice 1: Introduction to Java

Software Design (614G01015)

Eduardo Mosqueira Rey (Coordinador)

Department of Computer Science and Information Technology
Faculty of Computer Science



UNIVERSIDADE DA CORUÑA