IDE - Integrated Development Environment

# Introduction

Last semester you learned to program in Java. The tools you used were a text editor (KATE) and command line tools from the JDK (Java Development Kit) like *javac* and *java*. Remember all those *typos* and *compilation errors* you missed when you wrote the code? Those were a real pain. The good news is there is a way to avoid them and it is highly accessible.

This chapter is all about the developers best friend: the IDE. In this chapter you will learn what an IDE is, why you need it, how you can use it and why it will make your life a lot easier.

# What is an IDE?

Short for Integrated Development Environment, an IDE is a suite of tools brought together under the same roof to help the developer write code. Some of the tools that are part of almost all ides are: a smart text editor, a compiler, a runtime environment, an integrated terminal, testing platforms, code libraries, etc. Some of the best IDES are: Eclipse, Netbeans, Visual Studio, IntelliJ IDEA (and all other Jetbrains products), Anaconda, etc.

# IntelliJ IDEA

IntelliJ IDEA is a Java IDE built by Jetbrains. Jetbrains is a company that provides software that help developers work faster and better. Among their products we can find IDEs for the most popular programming languages.

Jetbrains created extensive and amazing [documentation](https://www.jetbrains.com/idea/documentation/) for all of their IDEs. The rest of this laboratory will show some features the IntelliJ provides, so that you will be able to write code easier and faster.

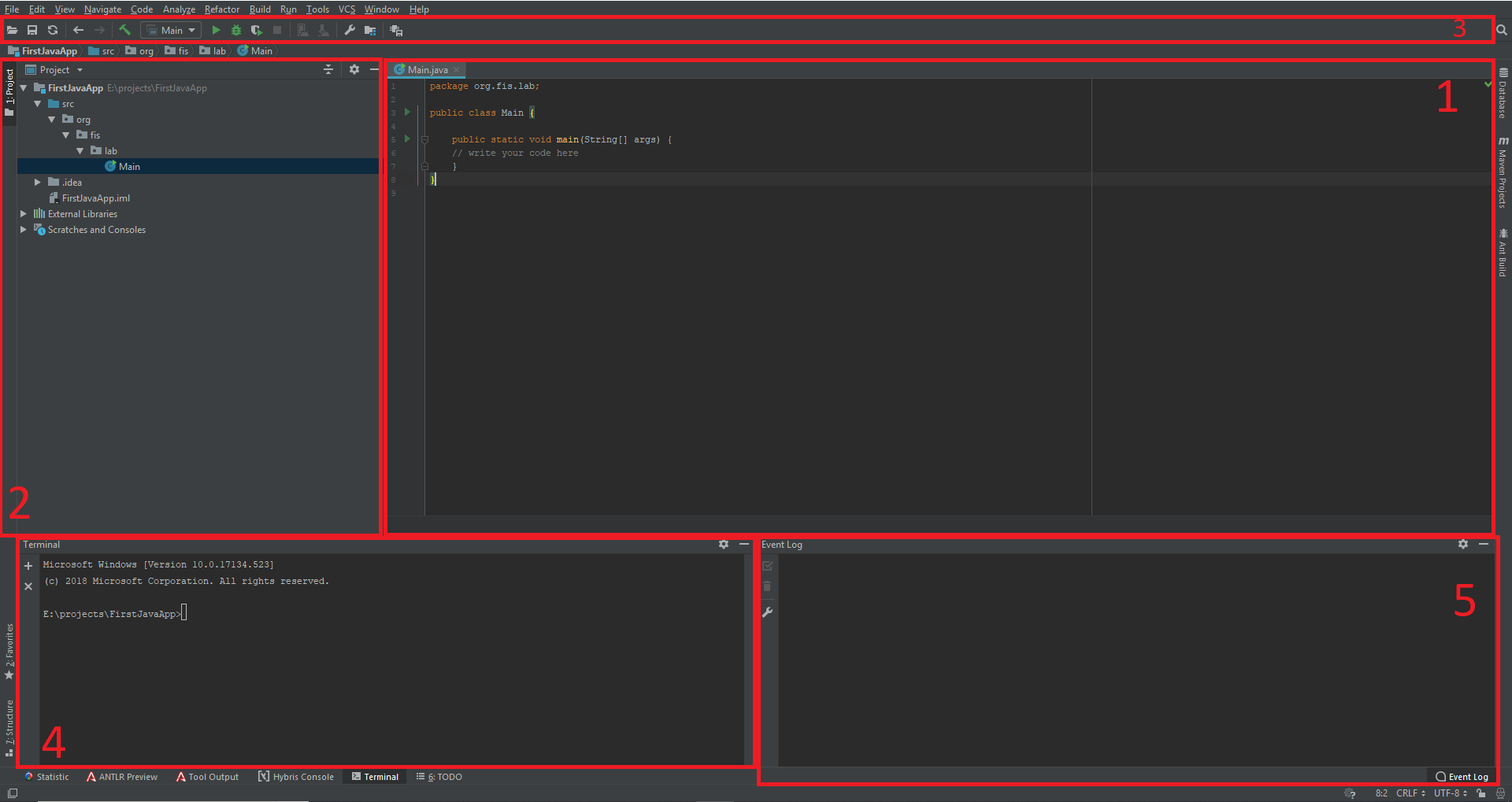
## Installation

Installing IntelliJ IDEA is straight forward. Just go to the IntelliJ IDEA [download page](https://www.jetbrains.com/idea/download/) and download the distribution that matches your operating system. You will see that you can choose from the free open-source version (IntelliJ IDEA Community) and the Ultimate edition. You can get access to the Ultimate edition using your UPT email account. Go to the Jetbrains [student license site](https://www.jetbrains.com/student/) and create your student account. Then download IntelliJ IDEA Ultimate and login with your account. In the lab you are going to use the Community version.

# Basic features

## The view

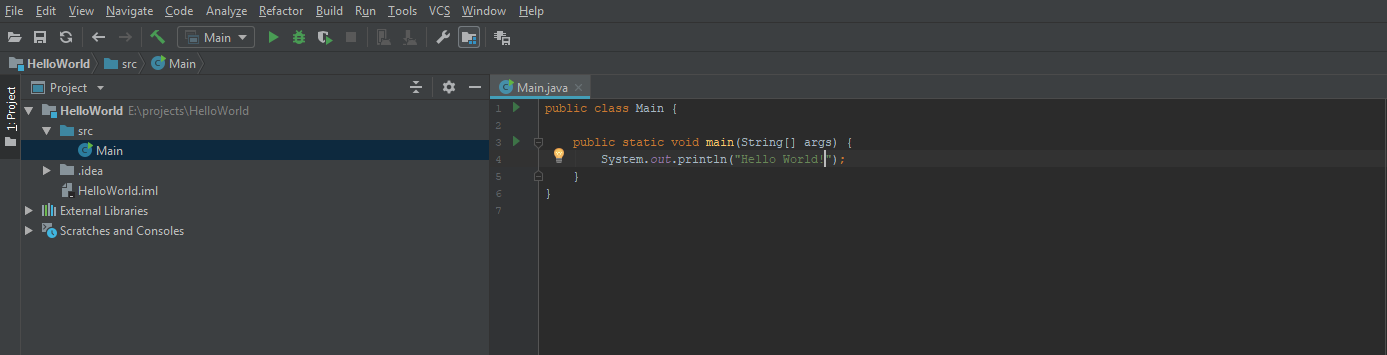
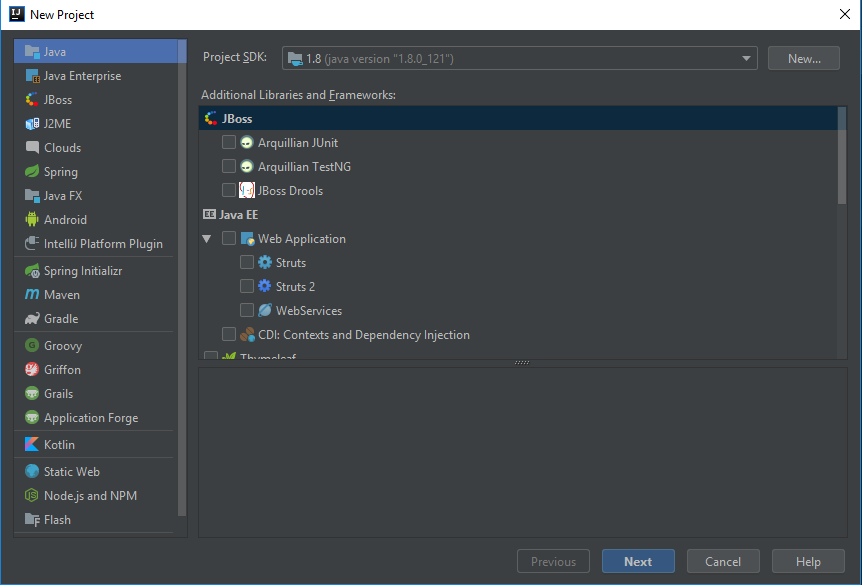
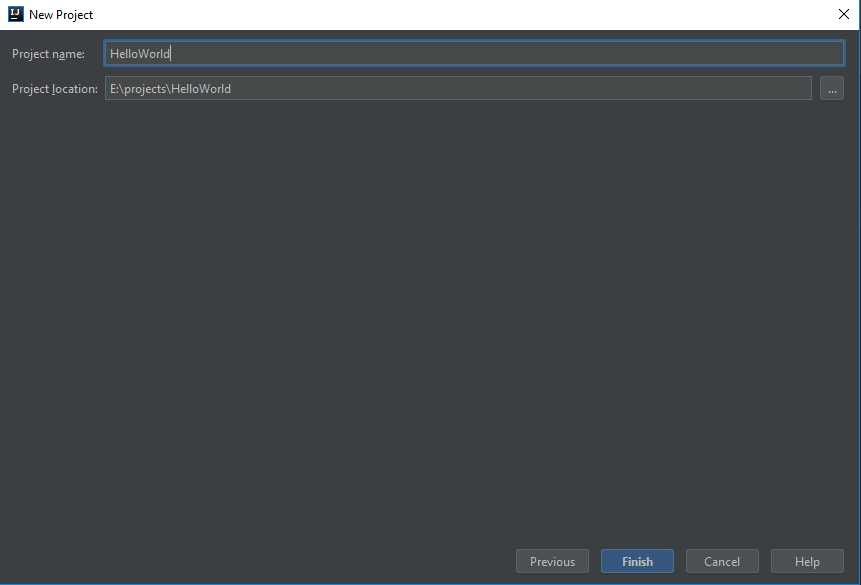
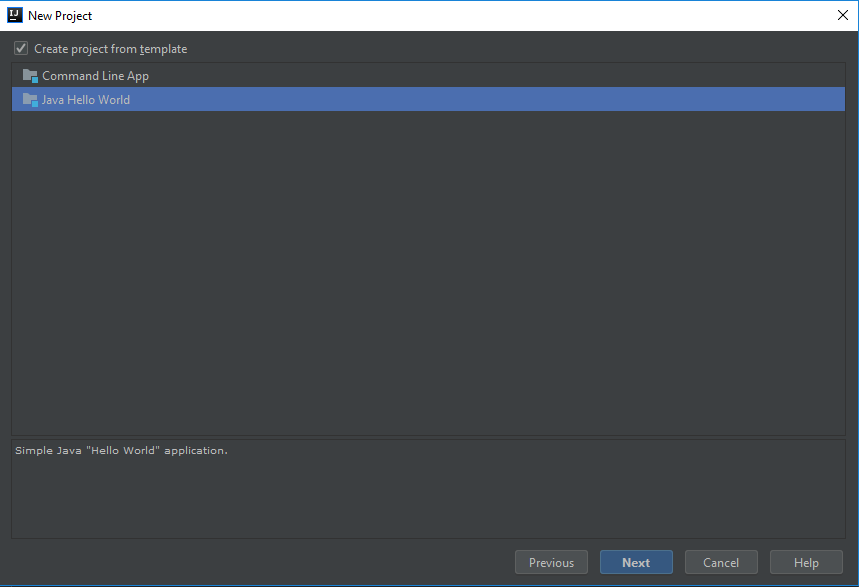
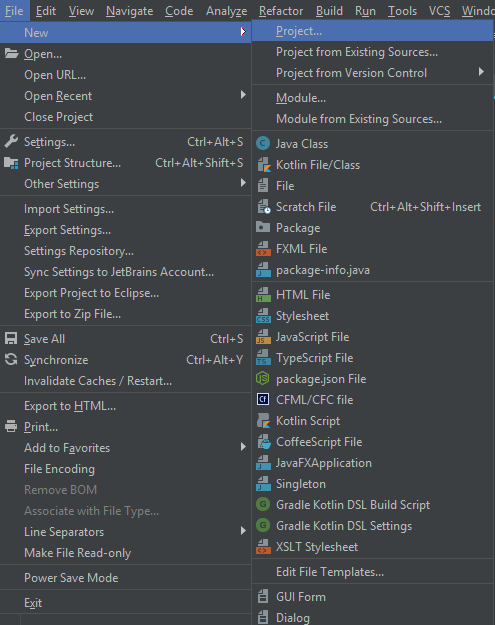
In this section we will describe the views and looks of IntelliJ. We will enumerate some of the most commonly used windows and show you how to show, hide or switch between them. So let’s get started.



The picture presents a lot of windows (if your IDE does not display all the views, make sure your IDE is setup to show Toolbar and Toolbuttons: in the main menu got to View and check *Toolbar* and *Toolbuttons)*.

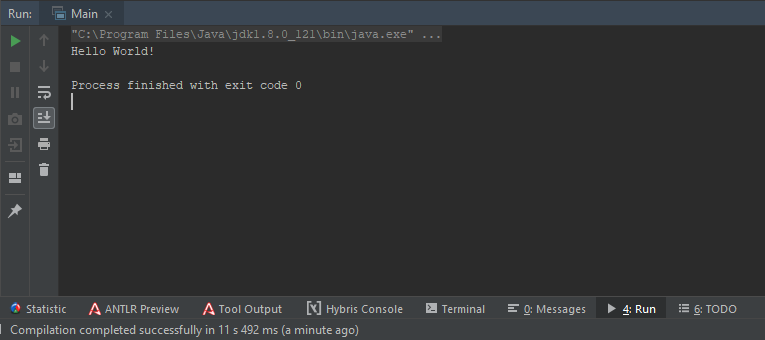
1. Text Editor View: this is where you actually write the code (or text). You can see at the top of this view the latest files that you opened.
2. This is the Project View: Here you can see the files of your project in a hierarchical view. You can click the settings Cog in this view an you will find different display options for your java packages and classes.
3. This is the Toolbar: Here you can find important actions buttons like the “Navigation Buttons” go to *Back* and *Forward* in the history of your file edits. Other important buttons are the *Run* or *Debug* buttons.
4. This is an integrated Terminal: Here you can write commands that you would normally run in your bash or command prompt.
5. This is the Event Log: Here IntelliJ IDEA will log any warnings, problems or info of the actions it is taking and you will always be up to date with the status of the actions you wanted to execute.

## Create a new Java project

In this section we will create a new Java Hello World Application and run it. To create a new Java Application, in the main Menu go to *File -> New -> Project…* . In the popup window select *Java* and click *Next*. Check the box (*Create project from template*), select *Java Hello World Application* and click *Next*.

Write *HelloWorld* as the project name and select a location for your project. After that your project is created and you can see a *Main* class with a *Hello World* printout.

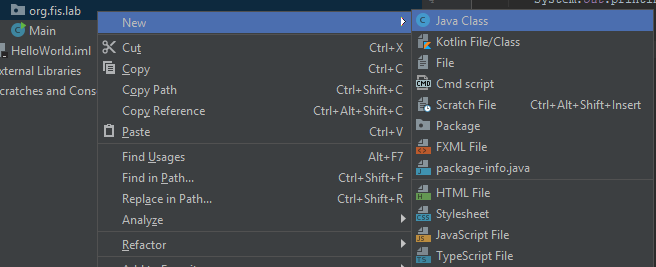
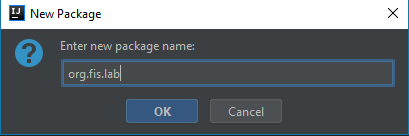
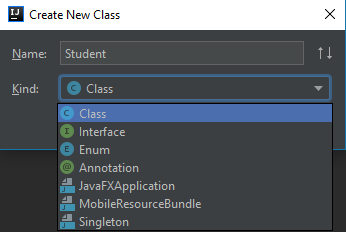
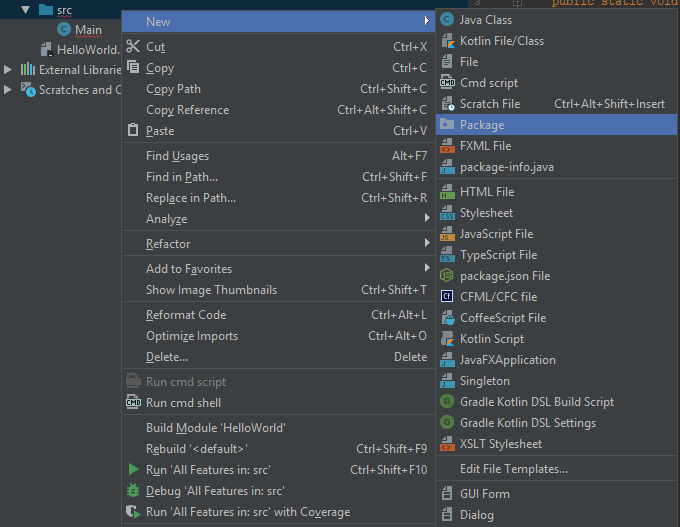
To run your project you have 3 options: The play sign to the left of the *main* method, the play sign to the left of the *Main* class, or the play sign on the Toolbar. Run the project and you will see a *Run* view will open and you will see *Hello World!* print out. You can notice that you can simply run the project (by pressing the *play sign*), debug it (by pressing the *bug symbol*), which we will talk about more in a later, or *Run with coverage*, which will show you how many of your lines were coverd during the run.

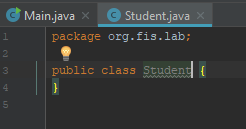
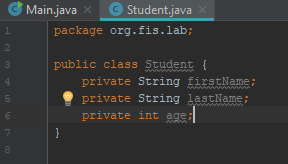


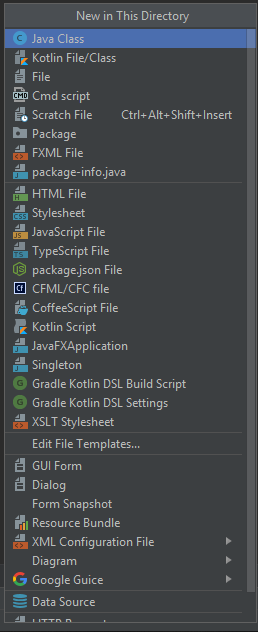
## Create packages and classes

In this section we will create some packages and classes. There are a couple of ways you can create a class in a package.

The mainstream way is to right-click the Sources folder *src* and click *New -> Package*, and type *org.fis.lab* as the package name. You will see that IntelliJ has created 3 imbricated packages for you.

Then right-click the lab package and click *New* -> *Class*. This will open a popup where you can create a new Java class. Type Student as the class name. 

The result will be something like this. Add 2 private String fields for first name and last name and an integer field called age.

A second way to create anything is to press *Ctrl + Alt + Insert*. This will open a dialog box where you can create most of the common files: 

The last and easiest way to create a class if you have the code would be to copy the code and paste it directly to the Sources folder (*src*).

**package** org.fis.lab;

**import** java.util.List;

**public class** Faculty {

**private** String **name**;

**private** String **adress**;

**private** List<Student> **students**;

}

Just copy the above snippet, click the *src* folder and *Ctrl + V* (or right-click -> *Paste*).

IntelliJ will automatically create The class and the package of the class (if it does not exist).

Until now we should already have 3 classes: *Main, org.fis.lab.Student* and *org.fis.lab.Faculty*. In the next sections we will write some more code in this classes by using the IDE to help us write faster and more efficiently.

## 

## Code generation

What we want to do now is expose the fields of the *Faculty* class through getters and setters. But it would be a lot to write them by hand. Luckily, IntelliJ is here to help. Just press *Alt + Insert* and you will see all the things that IntelliJ can generate for you. So go ahead and generate an empty and all-args constructor, getters, setters, equals(), hashCode() and toString() for both the *Faculty*  and the *Student* class.

## Slick moves:

* ***Alt + Enter***
* ***Ctrl + Space***
* Ctrl + W
* Shift / Ctrl + Alt + Arrows
* Ctrl (+ Shift + Alt) + N
* Ctrl + Shift + F / R
* Any other slick moves from [video](https://www.youtube.com/watch?v=eq3KiAH4IBI)

# Refactoring

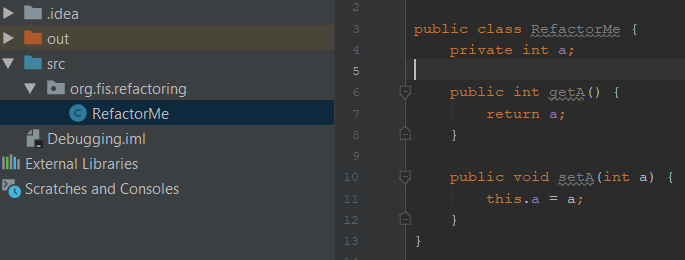
## Why refactoring?

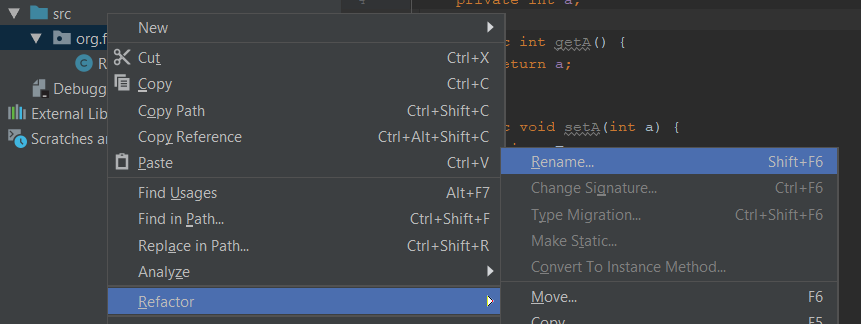
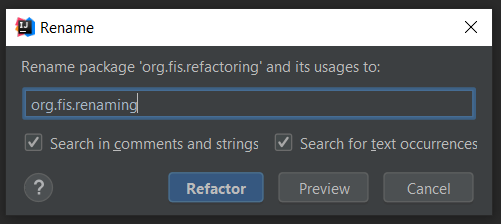
Refactoring is the process of restructuring the existent code in order to improve it but not change its behavior or add external functionality. The benefits of refactoring include: improvement of code readability, maintainability and extensibility.

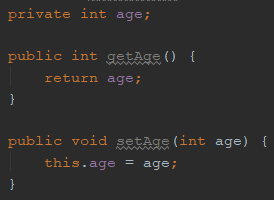
## Rename

Refactoring can be performed on every entity of a project and changes all the references to it throughout code. Renaming is a refactoring step that changes the entity name into a new one, that better defines its purpose.

Create the following project structure containing one class:



Let’s rename the package and change it to *org.fis.renaming*. Right-click on the package and select *Refactor -> Rename* or simply use the keyboard shortcut *Shift + F6*.

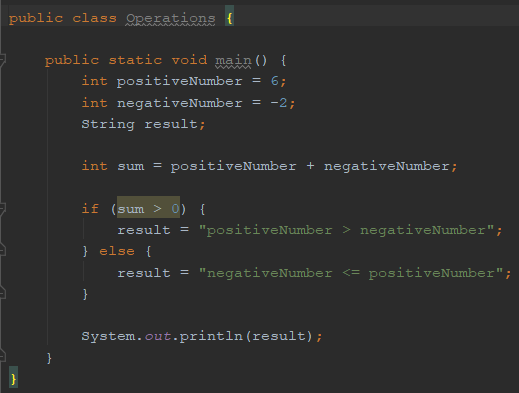
Also rename the class *RefactorMe* to *RenameMe*. In the *RenameMe* class you can see that there is a field named simply *a*. What does this field name stand for? We can only know that it has *int* type but it can mean anything: age, price, grade and many others. One of the best practices in naming conventions is to use meaningful names for variables and methods so that the name should reflect exactly the content. Let’s say that *a* stands for *age* so that’s why we need to rename the variable to *age*:

After doing so the getter and setter names are automatically changed.

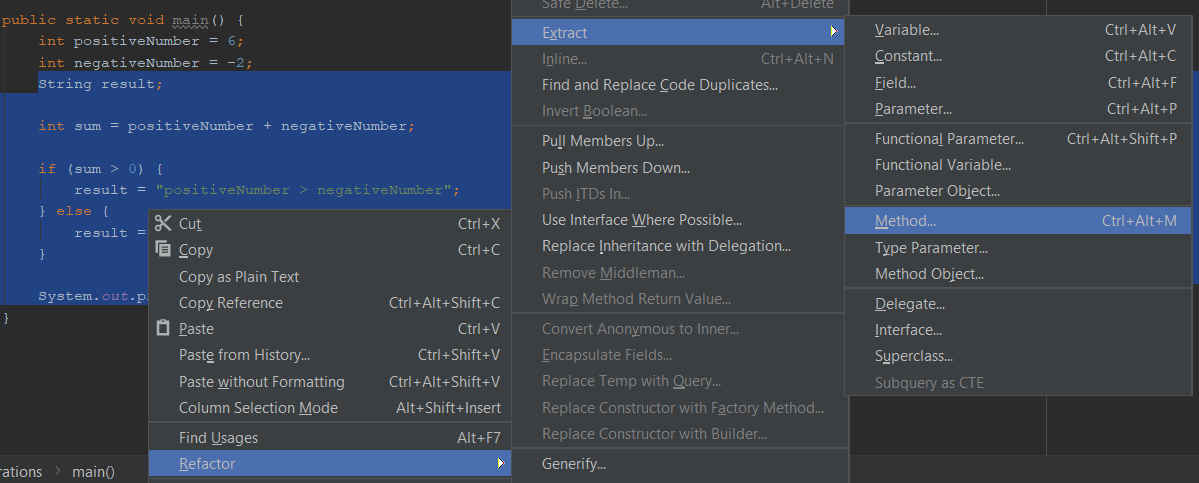
## Extract Method + Variable + Class

* Extract Method

The Extract Method refactoring lets you take a code fragment that can be grouped together, move it into a separated method and replace the old code with call to the method.

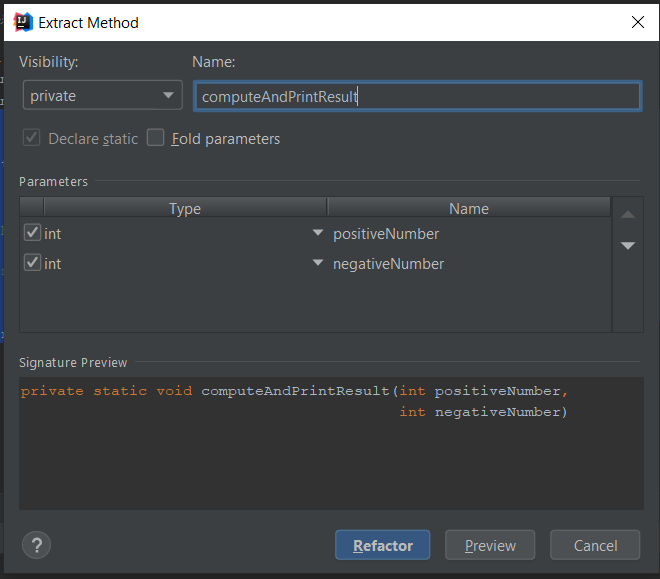
Having the *Operations* class where in the *main* method some mathematical operations are computed we can see that all the computational part can be extracted in a separate method.

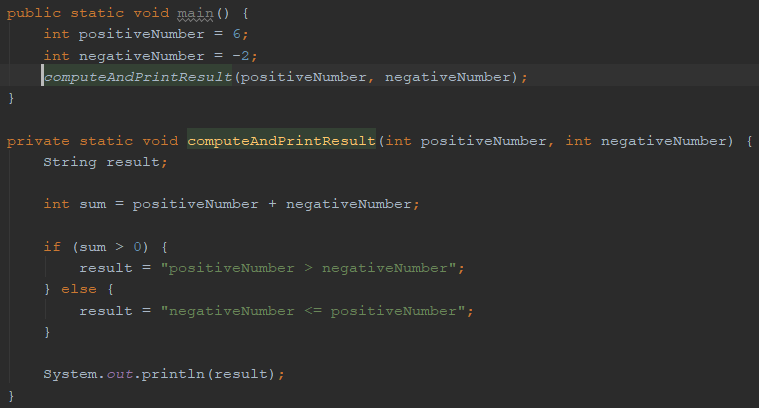
We select the entire code area which we want to extract and we can right-click on it and select: *Refactor -> Extract -> Method* or simply use the *Ctrl + Alt + M* shortcut.



A pop-up windows appears where we can name the method and change visibility and parameters.

After pressing the *Refactor* button our method is created and *main* method has a call to it. You can run it to see the results.

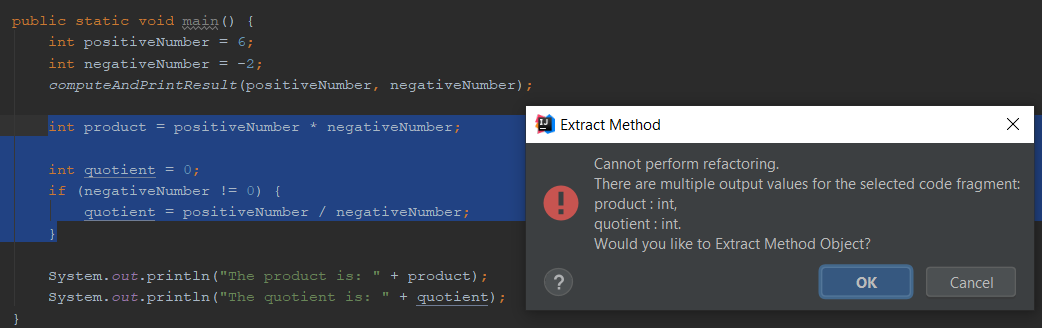




The Extract Method refactoring has the following limitations:

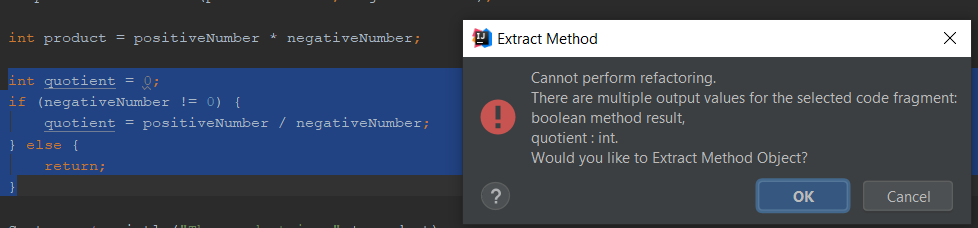
* Refactoring does not work with **multiple output values** in automatic mode. You have to change your code before applying the refactoring.

For example, if we add the following lines to the main method and decide to extract them in a separate one and after that showing the values for product and quotient, we will get an error, because we would have 2 output values for a method:



* Refactoring does not work for a code fragment which conditionally returns from the containing method and is not placed at the end of it.

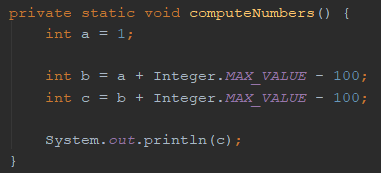
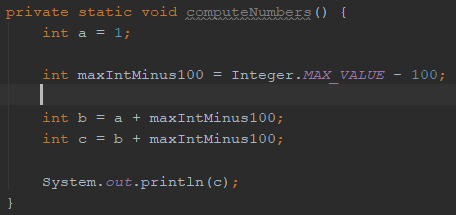
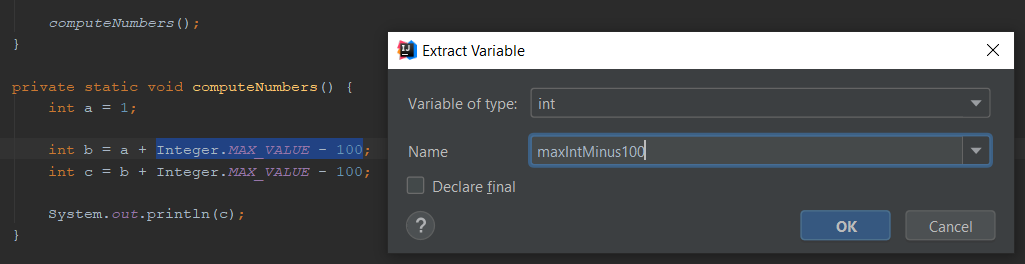
For example, on the same code snippet if we put an *else* statement with a *return;* and we want to extract that into a new method, we will get an error.



* Extract Variable

If you come across an expression that is hard to understand or it is duplicated in several places throughout you code, the Extract Variable refactoring can help you deal with those problems placing the result of such expression or its part into a separate variable that is less complex and easier to understand. Plus, it reduces the code duplication.

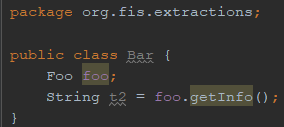
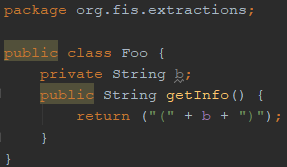
On the code snippet below you can see that there is duplicated the expression Integer.*MAX\_VALUE* - 100 so we can extract this as a variable and use it like that. We select the expression and press *Ctrl + Alt + V* to name our newly extracted variable.



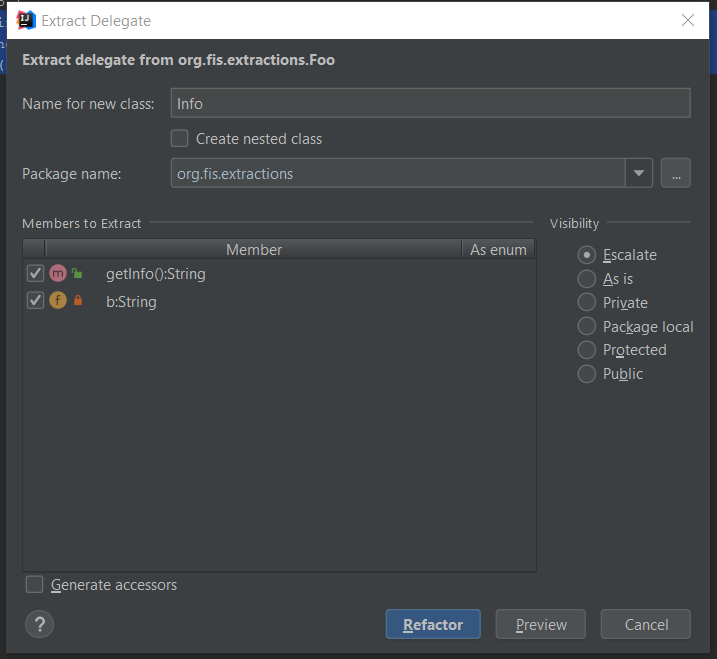
* Extract Class

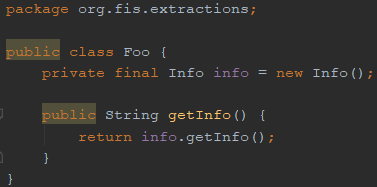
IntelliJ IDEA lets you use refactorings that extract fields, methods, and parameters into a new class. These refactorings are useful, when a class has grown too large and "does too many things". In such cases, it might be a good idea to split the class into smaller, more cohesive classes.

Consider having the following 2 classes, *Foo* and *Bar*:

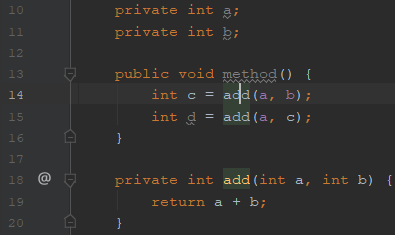


Let’s extract the lines inside *Foo* into a new class *Info* by selecting the lines and then right-click on them followed by: *Refactor -> Extract -> Delegate…*

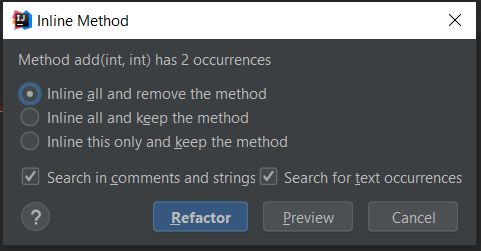


Now *Foo* has an *Info* object and *Info* contains its old field and method:

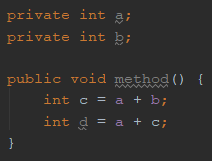
## Inline methods

The Inline refactoring lets you reverse the extract refactoring for method, constructor, parameter, superclass, anonymous class, and variable. Inline Method results in placing method's body into the body of its caller(s).

Consider the following lines inside a class:

We can get rid of the extra *add* method and simply make the sum inline. With the mouse placed on the *add* method call on line 14 and with *Ctrl + Alt + N* the *Inline Method* pop-up appears:

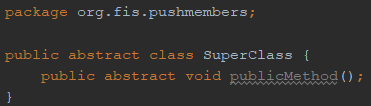
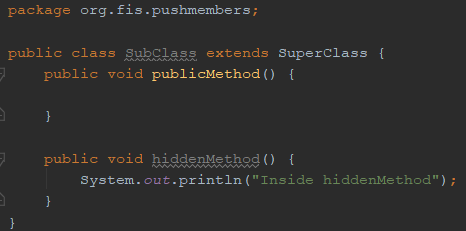
By clicking the *Refactor* button the *add* method is removed and the result is:



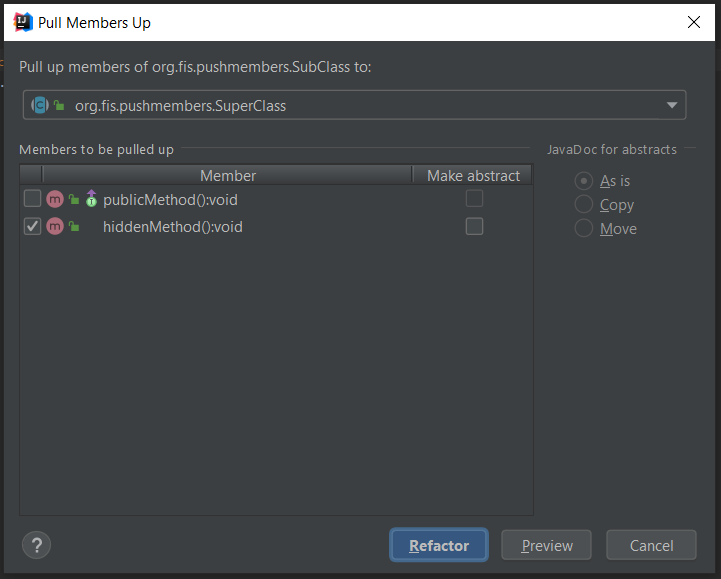
## Pull Members Up

The Pull Members Up refactoring lets you move fields if two classes have the same field, or move methods if your subclasses have methods performing similar work. This helps you to get rid of duplicate code. You can also move class members to a superclass or an interface, or interface to superinterface.

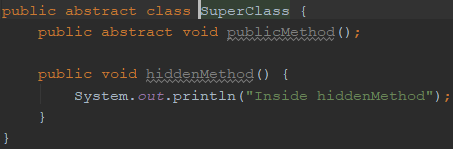
Having the following 2 classes: one parent abstract class *SuperClass* and a child class *SubClass*:



The *hiddenMethod* inside *SubClass* can be moved up in the superclass and also used in other future subclasses. The Pull Up operation for *hiddenMethod* is done by clicking *Refactor -> Pull Members Up...*



After the refactoring is performed, the *SuperClass* becomes:



## Push Members Down

The Push Members Down refactoring lets you clean up the class hierarchy by moving class members to a subclass or a subinterface. The members are then relocated into the direct subclasses/interfaces only. This improves internal class coherency.

This refactoring method is opposed to *Pull Mempers Up*. So we will use the same example as above, and push down the *hiddenMethod* into the subclass: *Refactor -> Push Members Down…* Now the structure of the classes should be exactly the initial one.

# Debugging

## Why debugging?

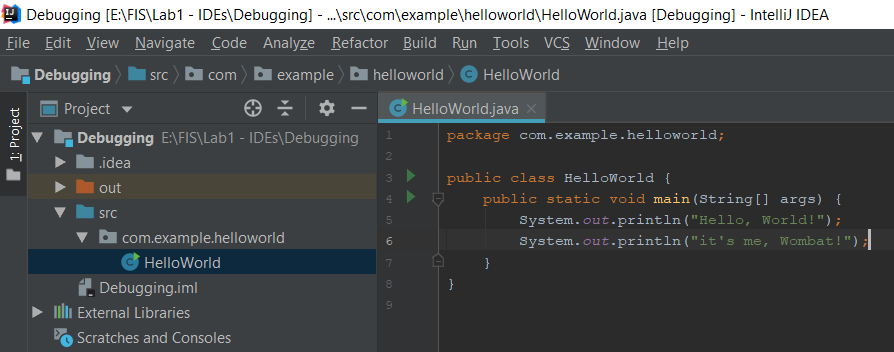
Besides writing the code, another important part in programming is testing and debugging - which means removing the bugs = the mistakes made when writing the code.



Considering the large part that debugging takes in the development cycle, everyone should know about the debugging techniques and try to master them in order to spend as little time as possible eliminating bugs.

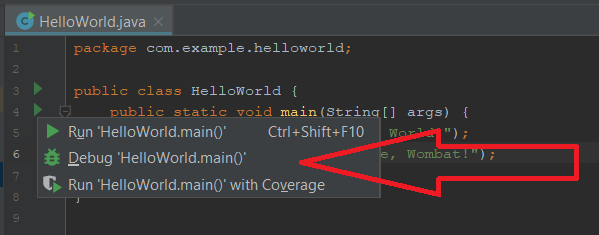
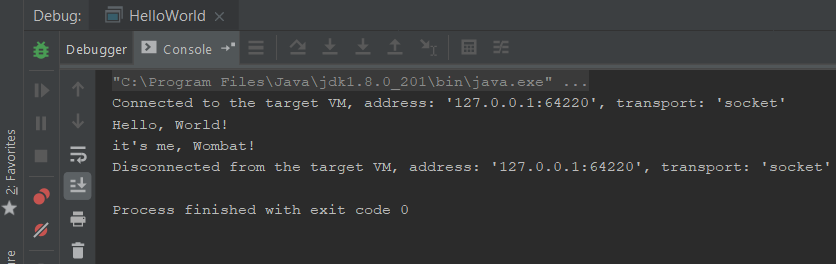
## How to start a debugging session

Consider having the following project generated by the Java HelloWorld template in IntelliJ:



We have the 2 lines in the main method that print 2 messages in console.

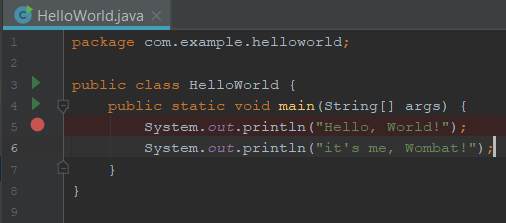
By running the Debug option: In the console we will have:



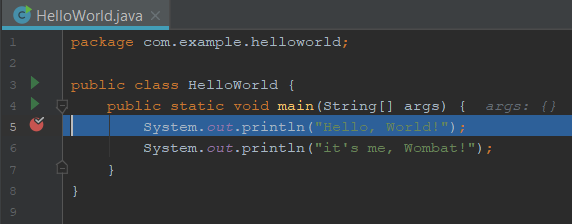
We can see that in this case the only difference in the Debug mode are the 2 extra lines in the console: *“Connected to the target VM...”* and *“Disconnected from the target VM...”*

## Working with breakpoints (conditionals / breakpoint panel)

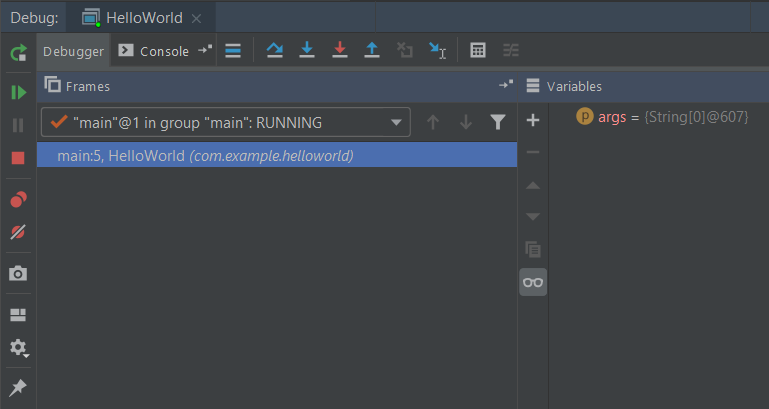
To suspend the execution of the application and inspect statements, you need to put breakpoints on them. This can be done by clicking one time in the left side on the line you want to put the breakpoint, in this case line number 5:



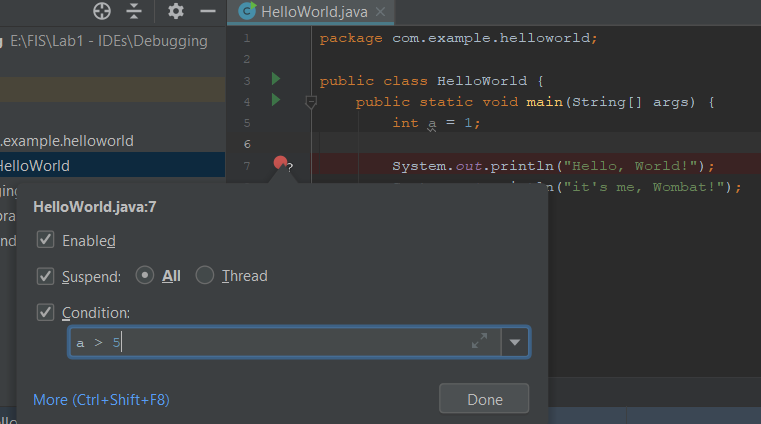
With the breakpoint set on line 5 if we run the main method in debug mode we can see that when the application reaches that line it gets suspended, the line is rendered blue and nothing is printed in the console:



In the lower part of the window now is visible the Debug tool window.



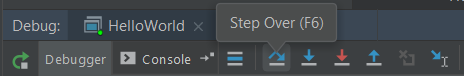
You can edit the properties of the breakpoint by right-clicking on it and for example setting a condition. We can have a dummy variable *a* for which we assign a value on which we will set the breakpoint condition:



With the condition set to: *a > 5* and our a being 1 when we will debug our main method we will see that the application is not suspended at the breakpoint. But if we change the condition, for example: *a< 5* the breakpoint now suspends the execution.

## Step into, step out, step over

If you have multiple breakpoints or if you want to navigate the code from one breakpoint you need to use the stepping toolbar that is shown on the top of the debugging console. The main actions that you will use are: Step Over, Step Into and Step Out.



A definition of what each one of them means:

* **Step Over**

Click this button to execute the program until the next line in the current method or file, skipping the methods referenced at the current execution point (if any). If the current line is the last one in the method, execution steps to the line executed right after this method.

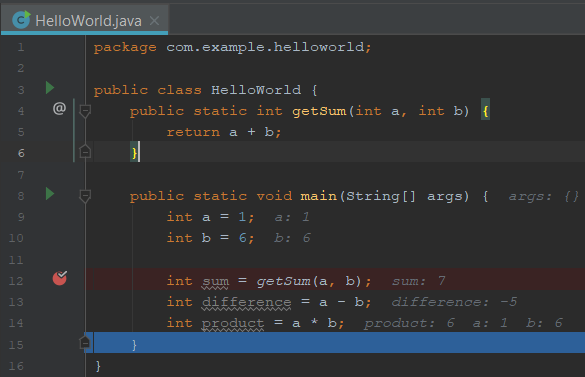
* **Step Into**

Click this button to have the debugger step into the method called at the current execution point.

* **Step Out**

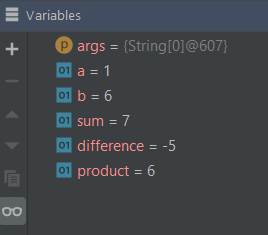
Click this button to have the debugger step out of the current method, to the line executed right after it.

Try them by yourself on the following short program:



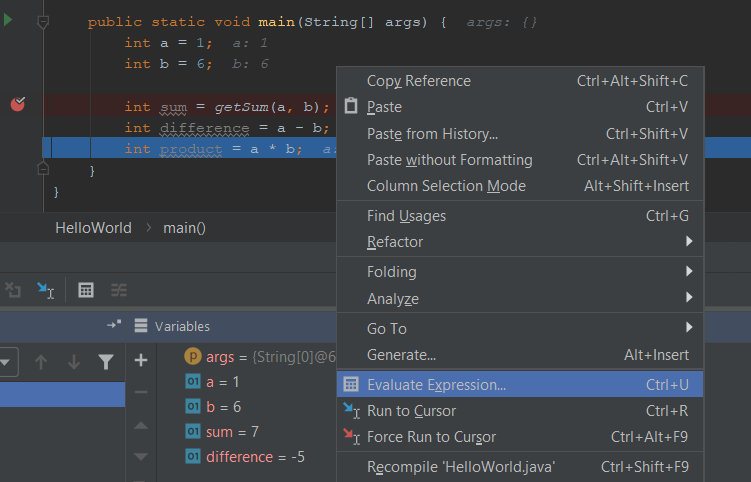
## Evaluate variables

IntelliJ makes it easy to evaluate variables because during the debugging time there are shown on each line, in italic and gray color, the values for each, easily to see in the example above.

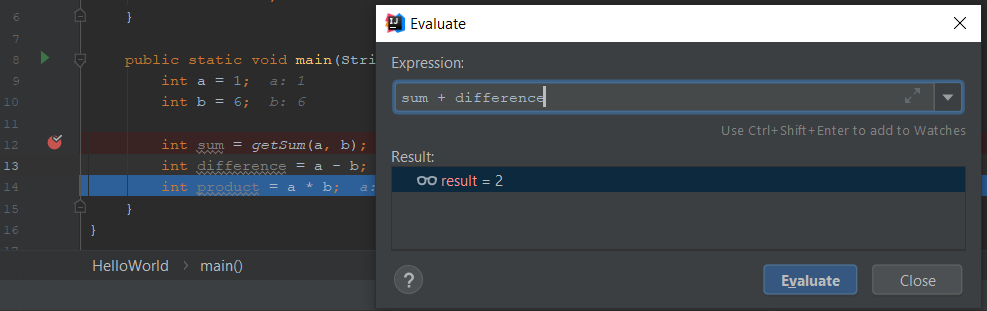
Also, it is easy to see these variables and their values in the Variables tab of the debugging view:

## Evaluate Expressions

You can evaluate expressions by right-clicking in the code editor and selecting the Evaluate Expression option.



In the pop-up for evaluation that comes after you can specify the expression you want to be evaluated, for example, when the debugger is at line 14 you can evaluate the expression: *sum + difference*:



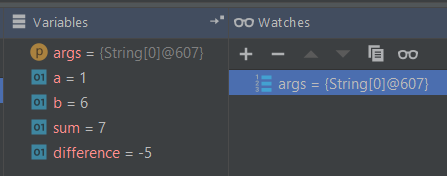
## Watches

In the Watches pane you can evaluate any number of variables or expressions in the context of the current stack frame. The values are updated with each step through the application, and become visible every time the application is suspended.

While the Evaluate Expression command on the context menu of the Variables pane enables you to see one expression at a time, the Watches pane shows multiple expressions that persist from one debug session to another, until you remove them.

You can create watches in this pane, in the Variables pane and even in the editor.

For example, we added to watch (using the + sign in the tab) the *args* variable that it also in the Variables tab:



# Hands-on exercise

In an information storage pseudo-system the following class hierarchies are considered:

Class **Document** has an attribute which is a sequence of strings (modelled either as an array, or as a list), and which corresponds to the whole text stored in the document. It also has a method suitable for displaying this text. The class also has a method *analyze* which returns a new sequence of strings, and whose implementation depends on the concrete type of document. The documents can be of two types:

1. **XML**. In this document, the information is represented as tags, each string in the sequence is either an opening tag (ex. "<name\_tag>"), either a string, or an end tag (ex. "</name\_tag>"). The *analyze* method returns a sequence of strings obtained by removing the beginning and end tags. The method suitable for displaying the text will return “XML” followed by the whole text of the document (including the tags). Example, assume a document with the follwoing content: <tag1> value1 </tag1> <tag2> </tag2> <tag3> value2 </tag3>

The *analyze* method will return for this document the sequence: value1, value2

It is taken as a given that the initial string sequence of the document will always correspond to the described format.

2. **JSON**. In this document, the elements of the string sequence are either names (like “name”), or a string value. The *analyze* method returns a sequence of strings obtained by removing all the names from the initial sequence. The method suitable for displaying the text will return “JSON” followed by the whole text of the document (including the names). Example, assume a document with this content: name1: value1 name2: value2. After the name elements there is a ‘:’ character, and for the given example the *analyze* method returns: value1, value2.

It is taken as a given that the initial string sequence of the document will always correspond to the described format.

In this system any document can be processed using a **Processor**. The following types of processors are considered:

1. **SearchProcessor**. This processor has a string attribute which is initialized via the constructor, and which corresponds to the text that will be searched for. The *process* method receives as parameter a sequence of documents and works in the following way: for each text obtained by analyzing a document (so with no tags or names considered) all occurences of the search query (the searched for string) are counted. The method returns the total number of occurences of the search query in all documents from the processed document sequence.

2. **ComposedProcessor**. This processor contains a sequence of processors (search processors or other composed processors). The *process* method receives as parameter a sequence of documents and works in the following way: it returns the sum of the results obtained by applying each processor from its processor sequence on all documents received as parameter by the *process* method.

Whenever it is needed, the sequences are initialized and/or populated either via the constructor or by using a suitable add element method.

**Requirements:**

- Implement the classes described above together with any other needed classes, by working in IntelliJ IDEA (mandatory). The code will be split in the following packages:

* *loose.oose.sef.documents* for the classes in the document class hierarchy;
* *loose.oose.sef.processors* for the classes in the processor class hierachy;
* *loose.oose.sef* (the root package for this project) will contain the *Main* class.

- Generate automatically the setter/getter method pairs for the attributes of class **Document** and the display method (*toString*)

- Test the correct functionality of the above system by implementing a *Main* class which:

1. Instantiates at least 2 documents (one of each type) based on the texts received as model
2. Instantiates at least 3 search processors, 1 composed processor that contains the first 2 search processor and 1 composed processor that contain the previous composed processor and the third search processor
3. Calls the *process* method on the last composed processor with a sequence of documents containing the objects instantiated at 1)
4. Debug the call to the ***process*** method.

- Refactor automatically the name of the string attribute of class **SearchProcessor**

# Conclusion

As we hope this document has proven, IDEs are a very important tool in software development. In further lectures we will show the full power that the IDE can provide through integrations with different technologies like Git, Maven, Gradle, Youtrack, etc. So stay tuned to learn more about features that can ease your life as you enter the world of a software developer.