

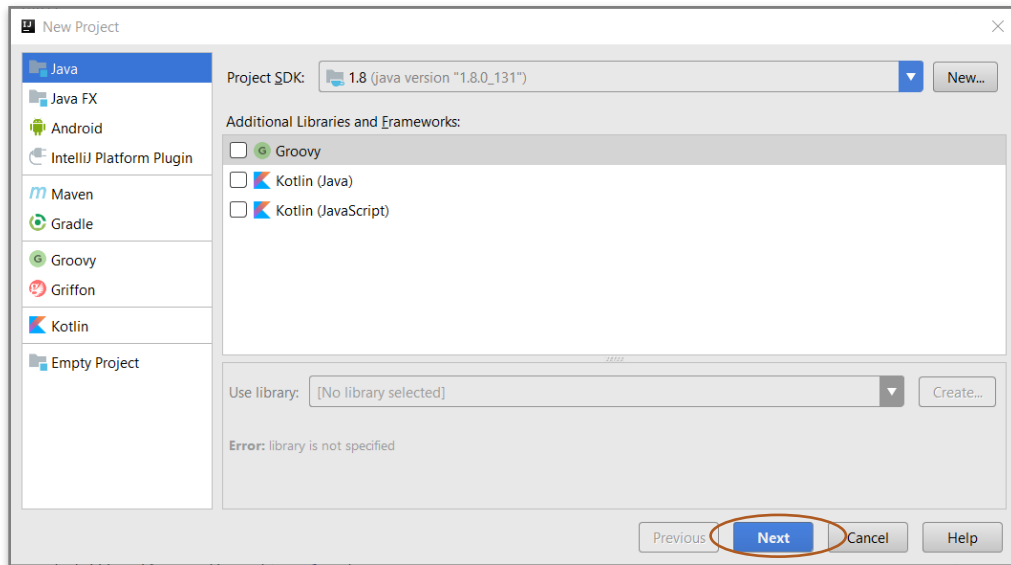
IntelliJ IDEA Tutorial

Contents

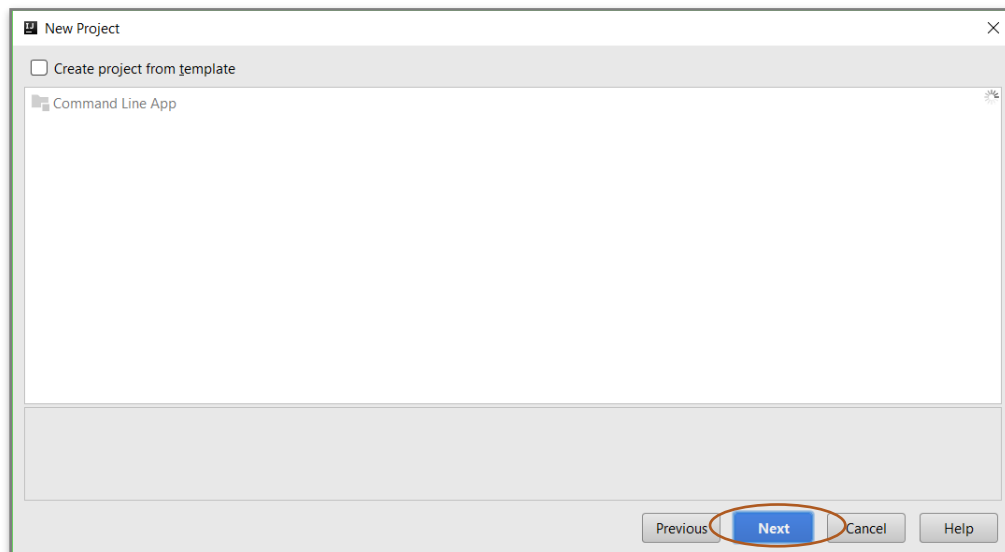
| | |
|-----------------------------------|----|
| Creating New Java Project | 2 |
| Opening A Project | 4 |
| Creating New Java Class..... | 5 |
| Running Your Application | 6 |
| Debugging Your Application | 10 |
| Adding Libraries | 13 |
| Testing Your Code | 20 |
| Running The Inspection Tool | 30 |

Creating New Java Project

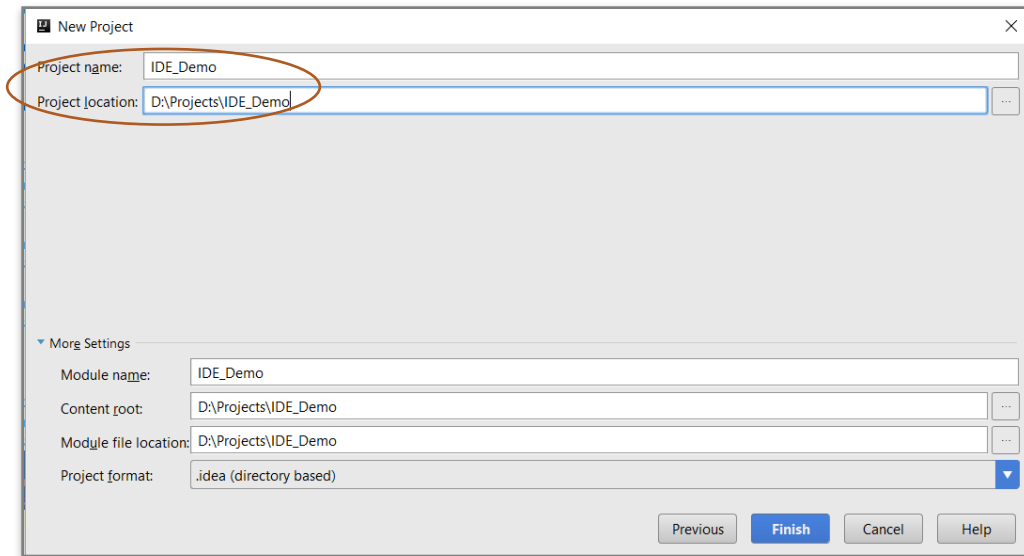
Click on **File** -> **New** -> **Project...** to open the New Project dialog box.



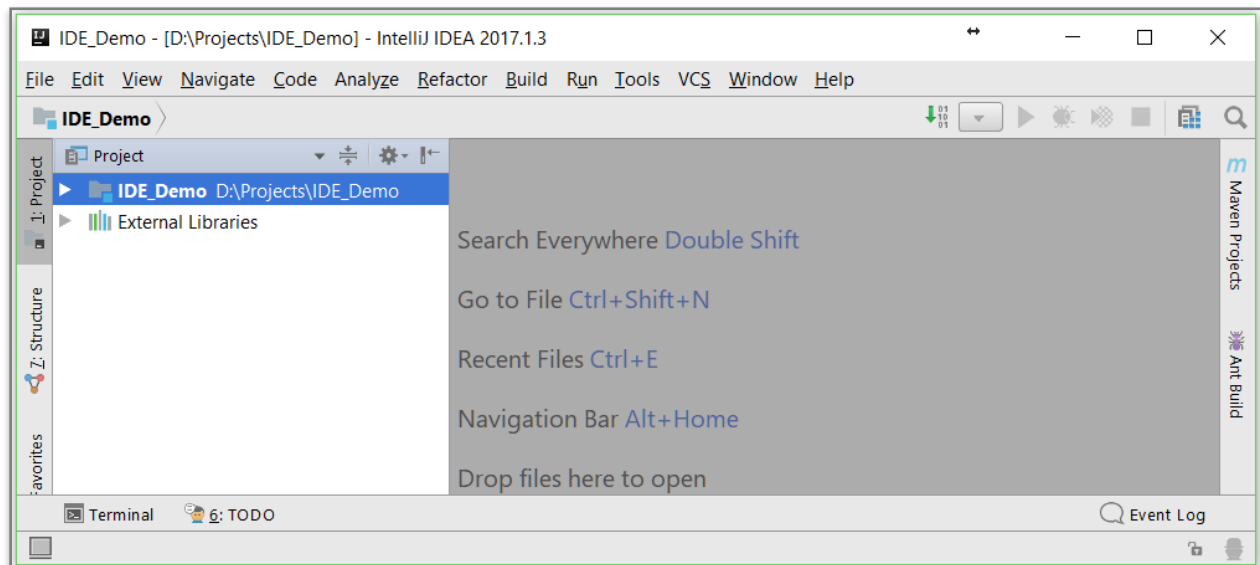
Click on **Next**



Pick a name for the project and select a directory for storing the project files, then click on **Finish**.

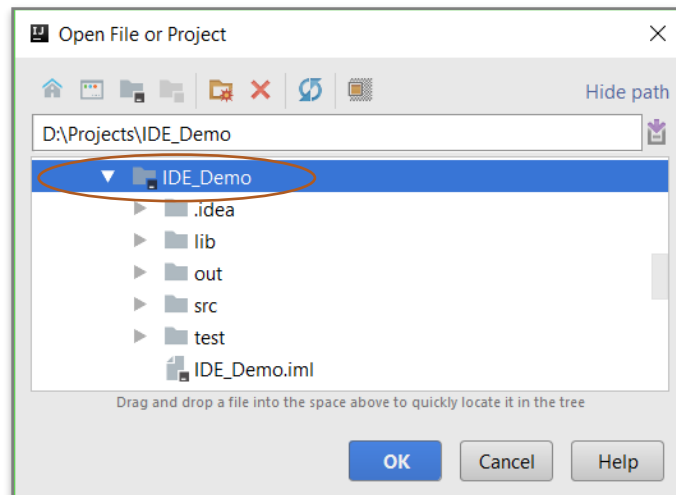


A new Java project will be created for you and opened in the IDE.



Opening A Project

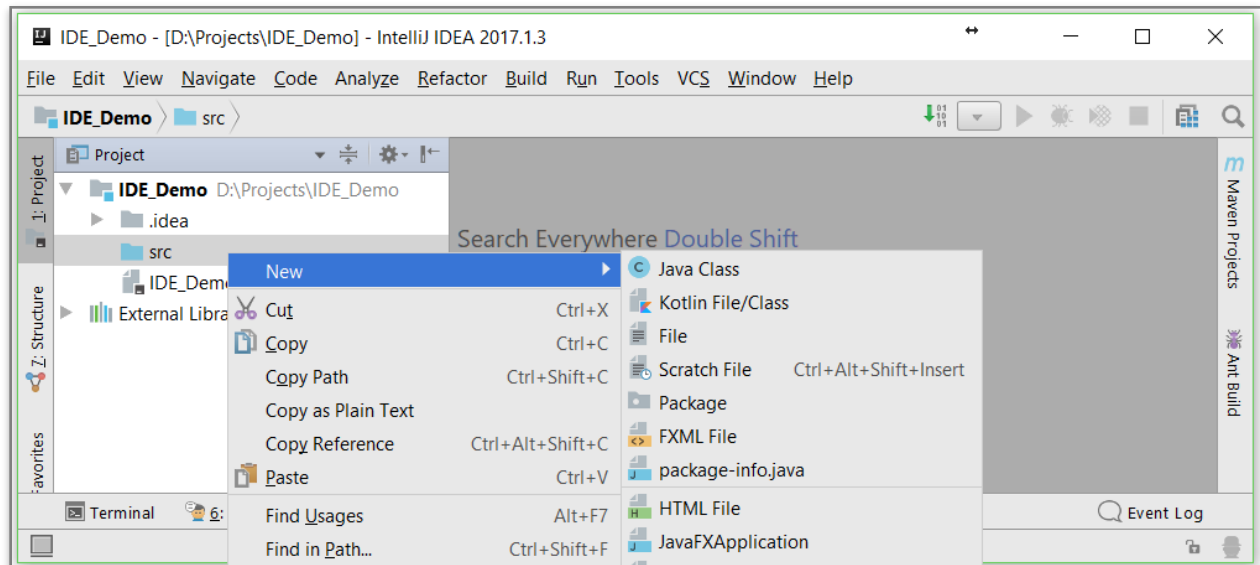
To open a previously saved IntelliJ IDEA Java project, click on File -> Open... Browse to and select the root folder of the project (i.e., the folder containing the .iml file of the project) you want to open, then click on OK.



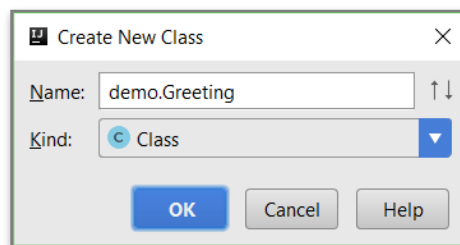
Note: Clicking on OK when the project file (with file extension name .iml) is selected will open the project file itself, not the whole project.

Creating New Java Class

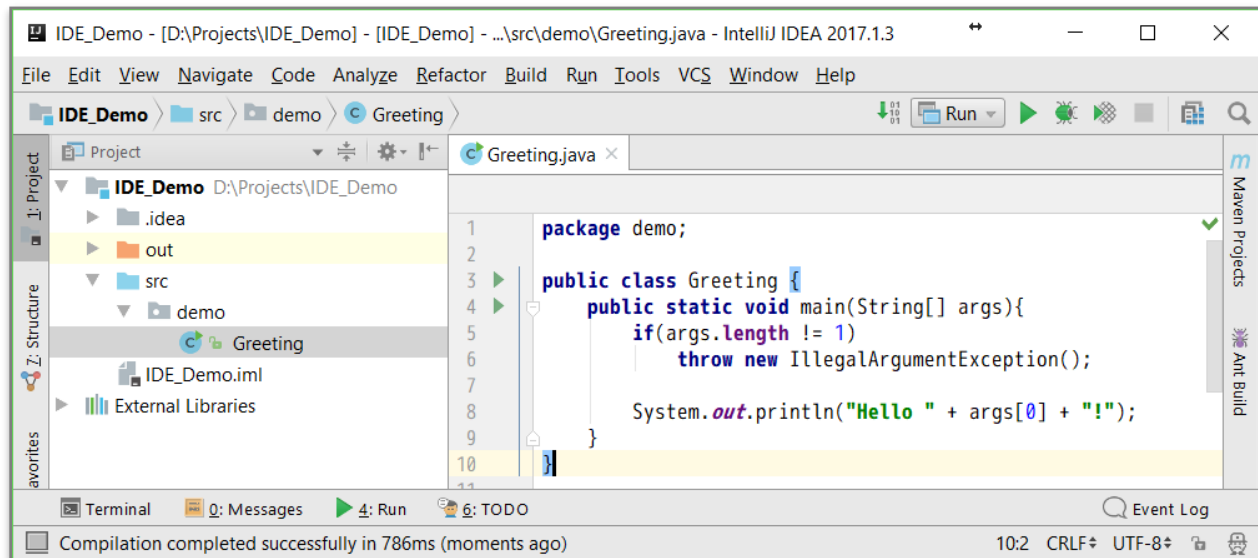
Right click on folder `src` of your project, then click on **New** -> **Java Class**.



Give the fully qualified name of your class. According to Java naming convention, package names should be in lower case and class names in upper case.

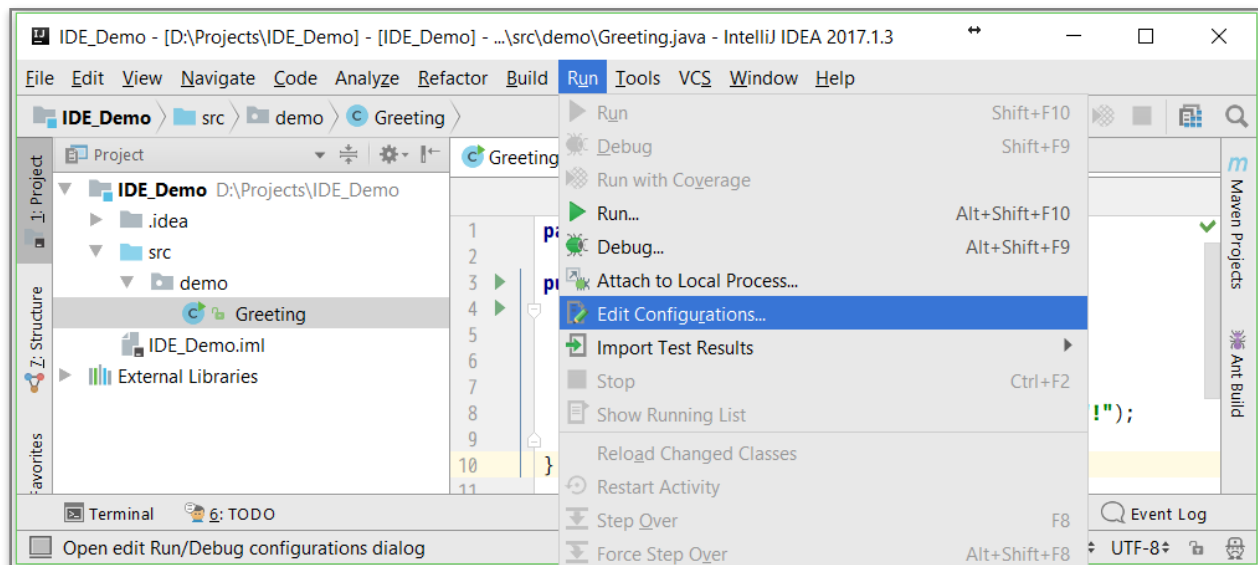


After clicking on OK, the Java class and corresponding package structure will be created for you. Double click on the class name to open the source file of the class in the Editor. Here our program expects exactly one argument and will print some greeting message.

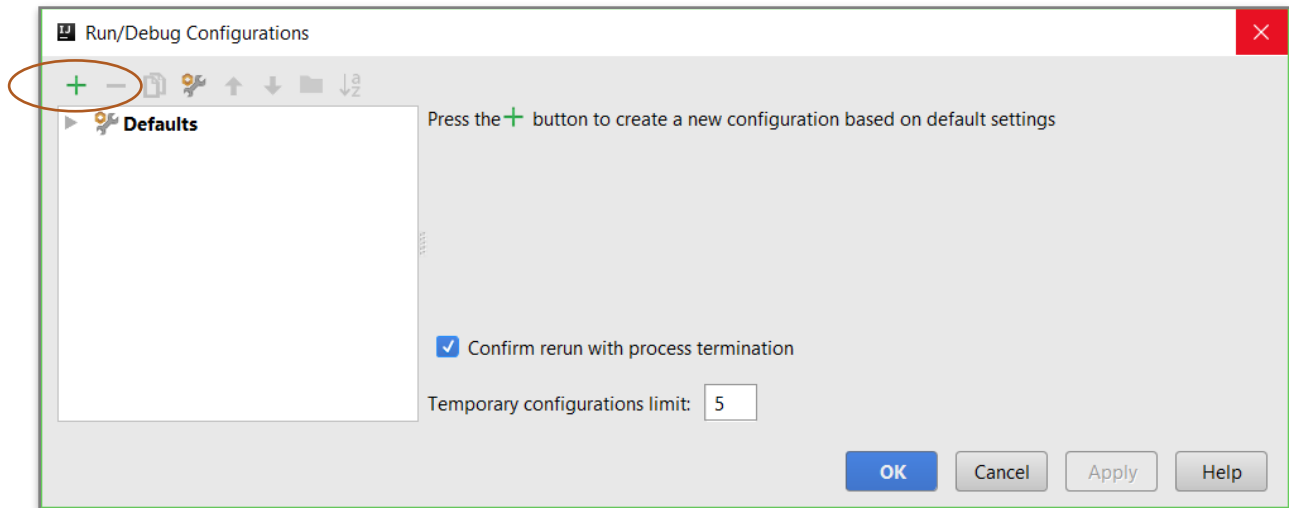


Running Your Application

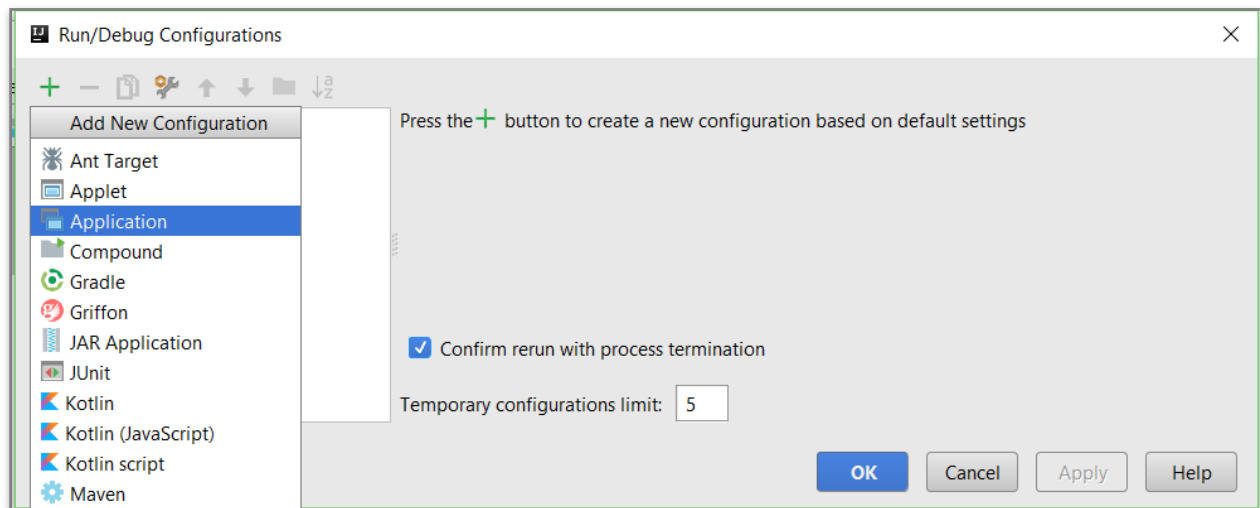
Before you can run the application, you first need to have the right configuration to be used for launching the application. To create a configuration, click on Run → Edit Configurations...



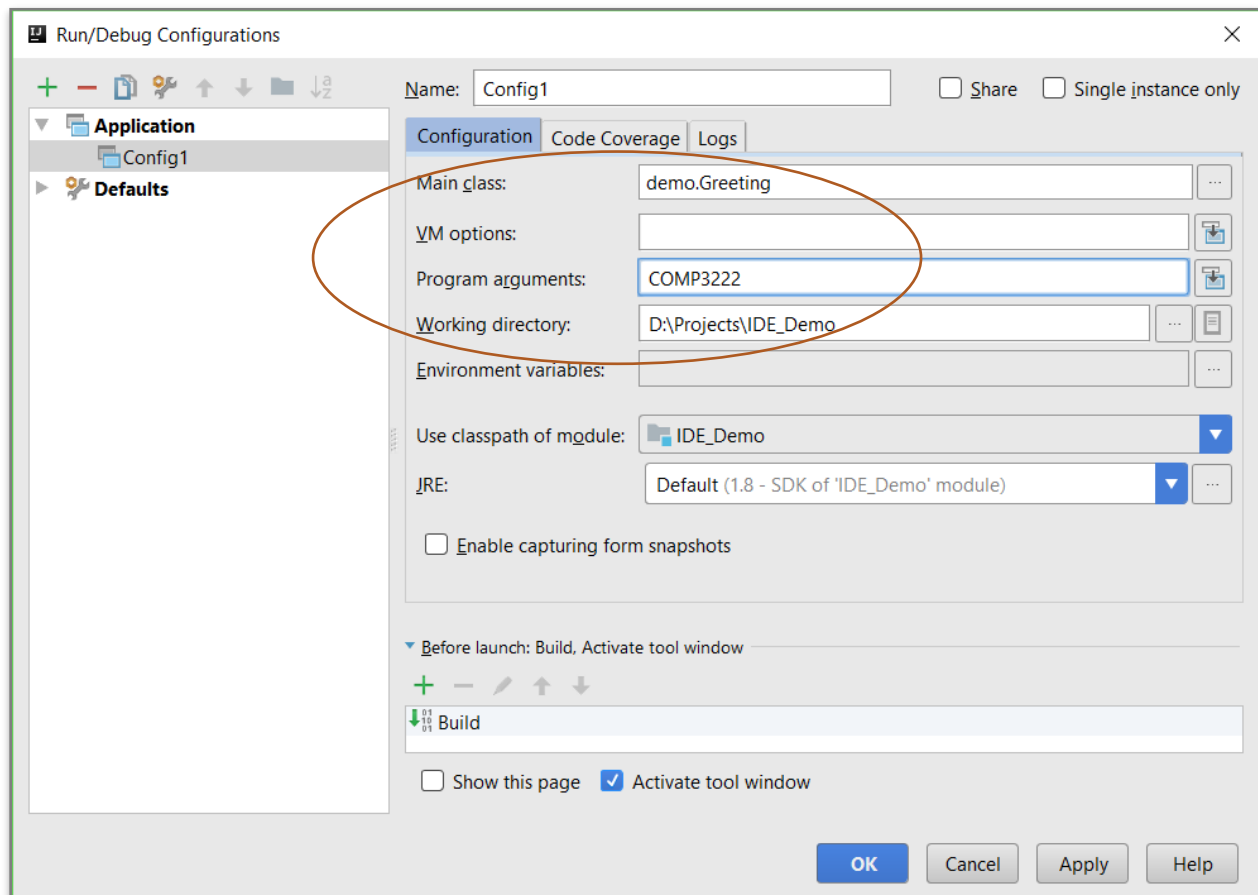
The Run/Debug Configurations dialog will open.



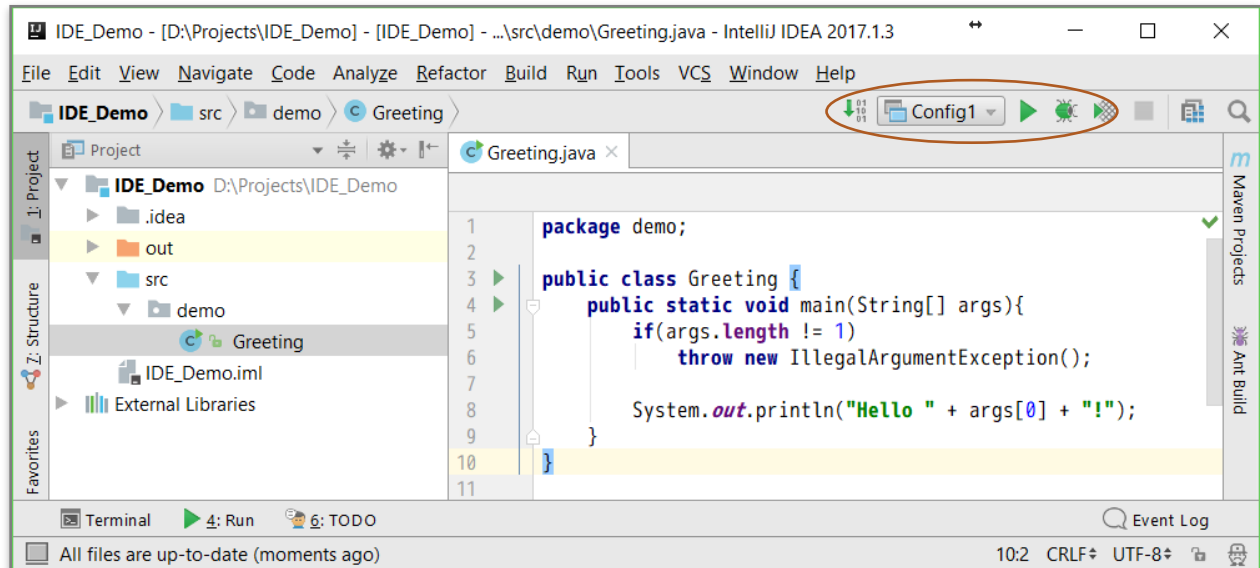
Click on the + button and then select the type of configuration you want to add. Here we want to run the program as an Java application, so we choose Application from the list.



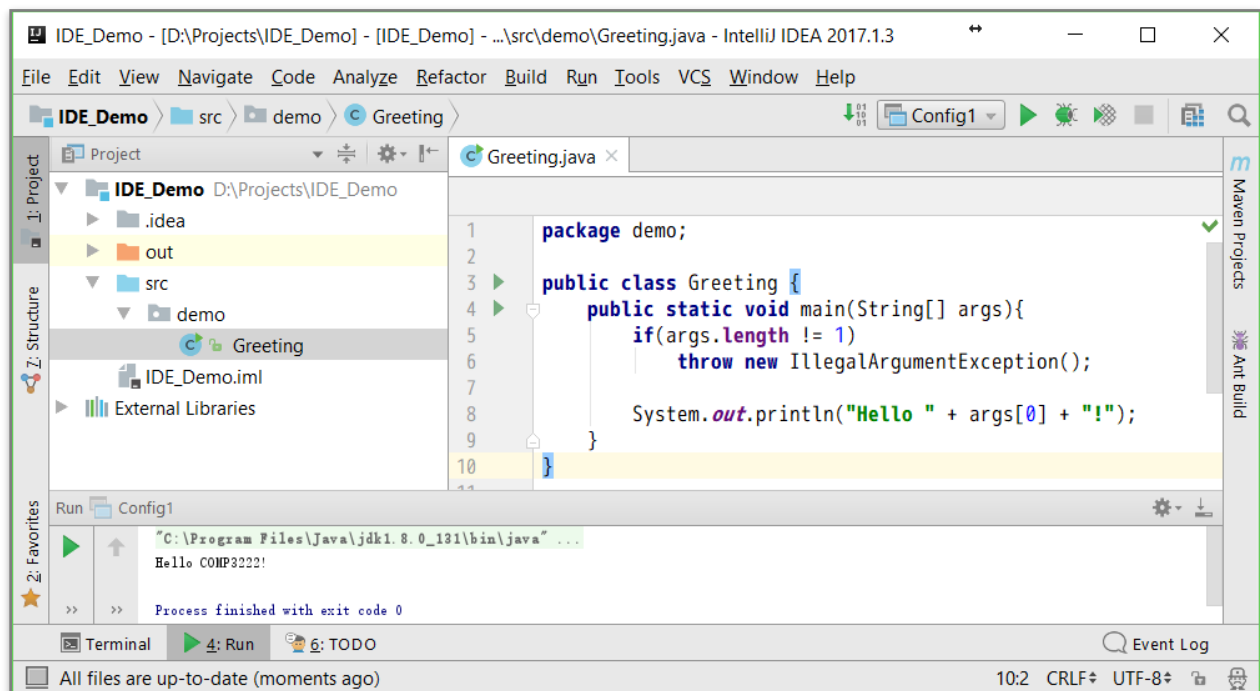
Name your configuration, input the **Main class** for the application (execution will start from the main method of the **Main class**), and provide all the **Program arguments**, if needed.



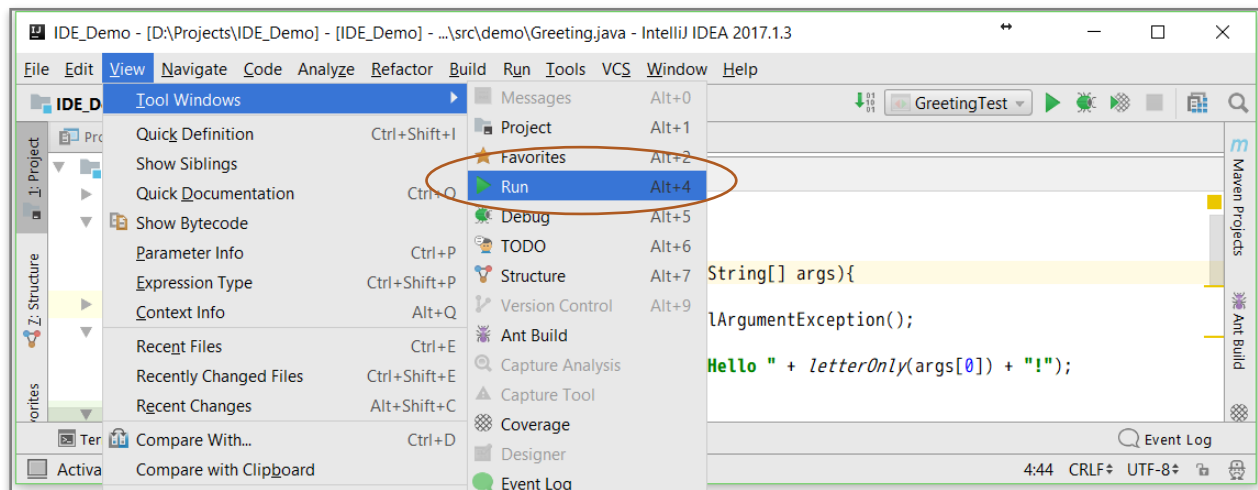
After clicking on OK, you will see the configuration appears in the toolbar. Next to it are the buttons for running/debugging the application.



Click on the Run button to launch the application. The output will be shown in the Run tool window.

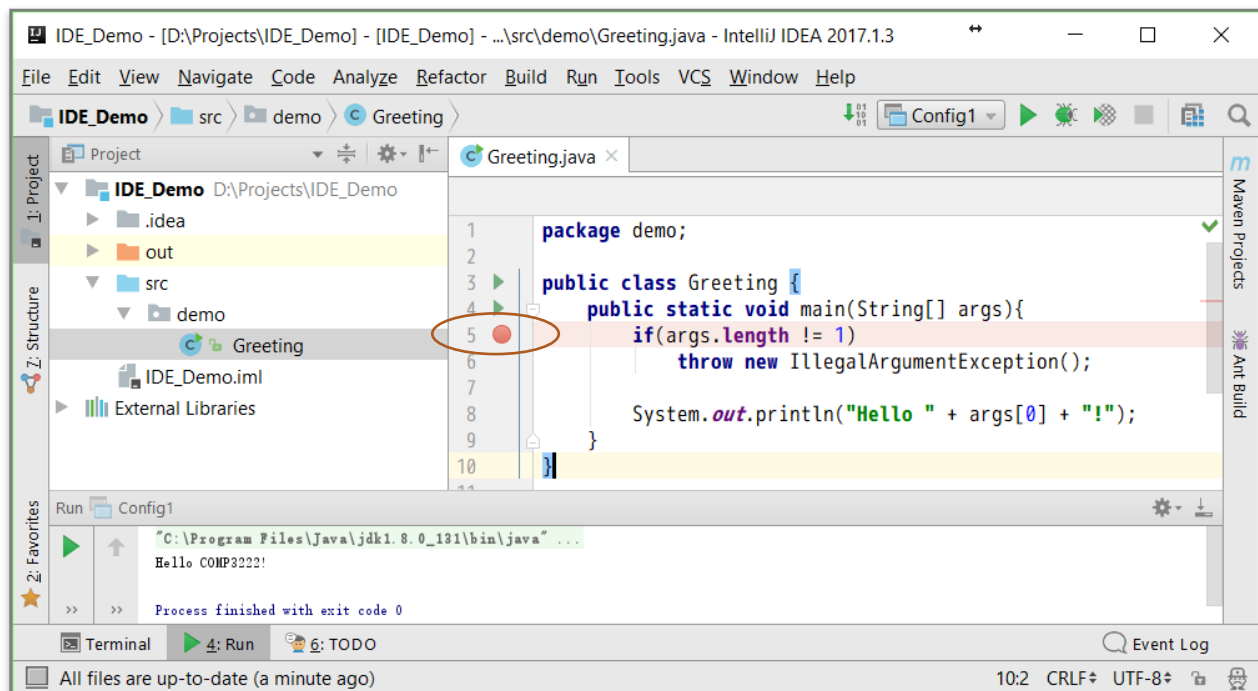


If the tool window is not opened automatically, you can manually open it by clicking on View -> Tool Windows -> Run.

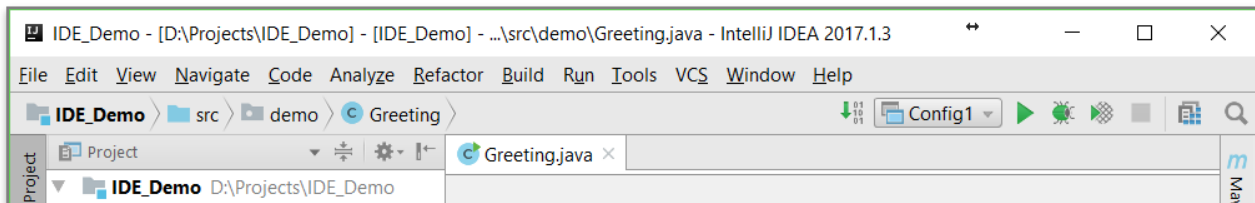
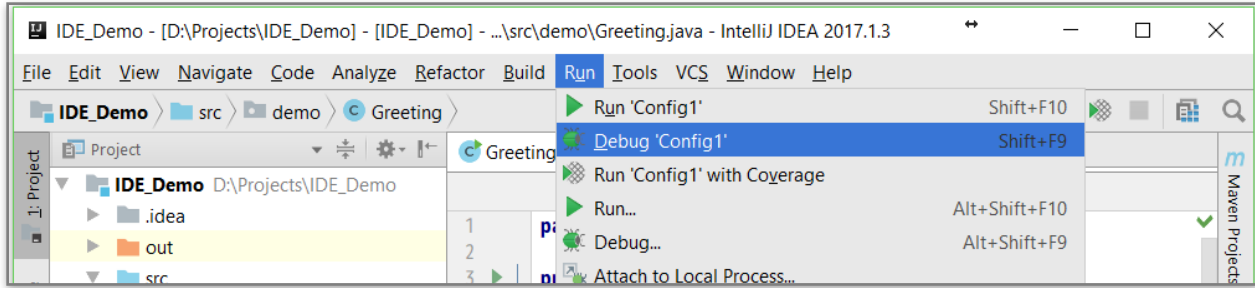


Debugging Your Application

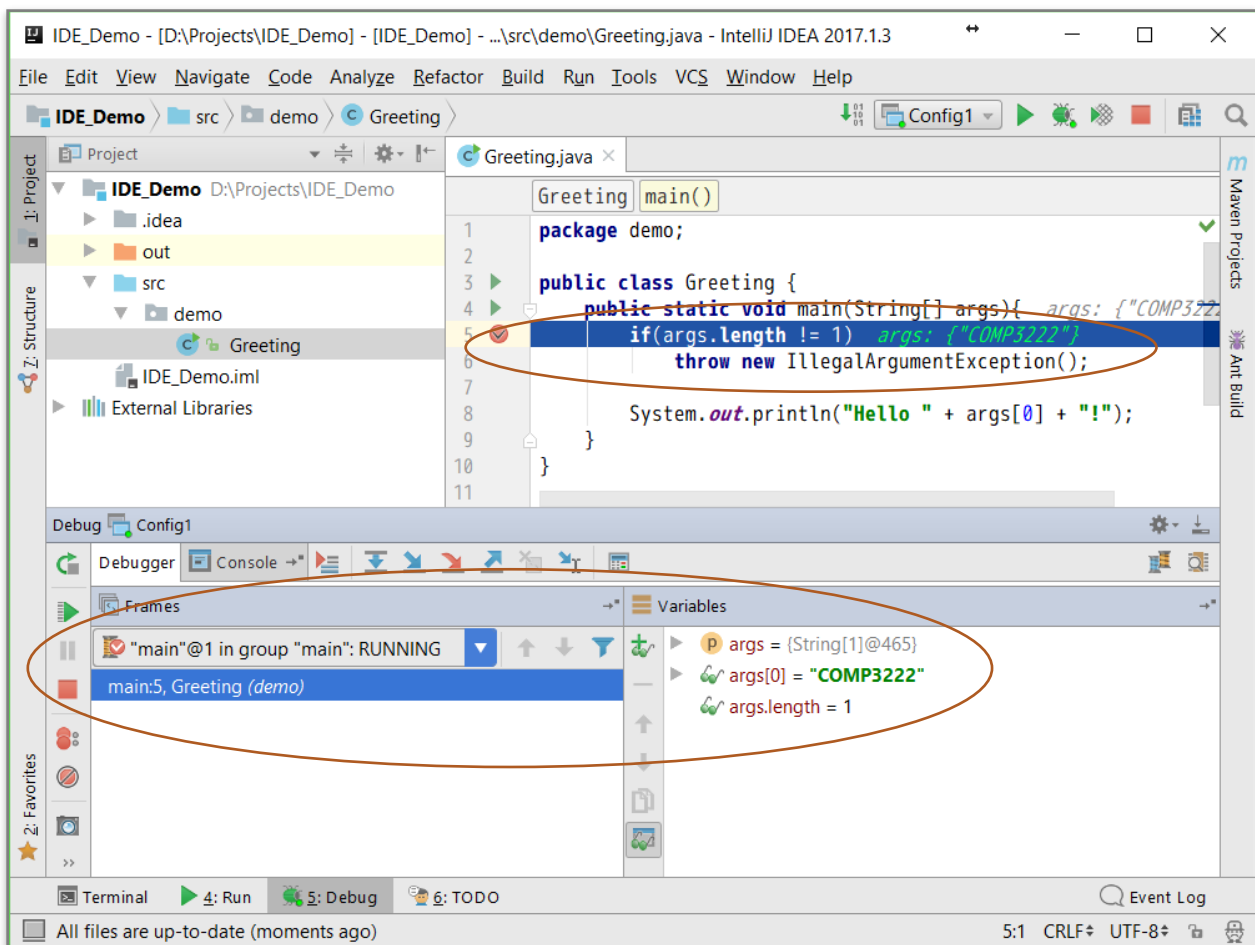
You can click before an executable line to toggle the breakpoint on that line.



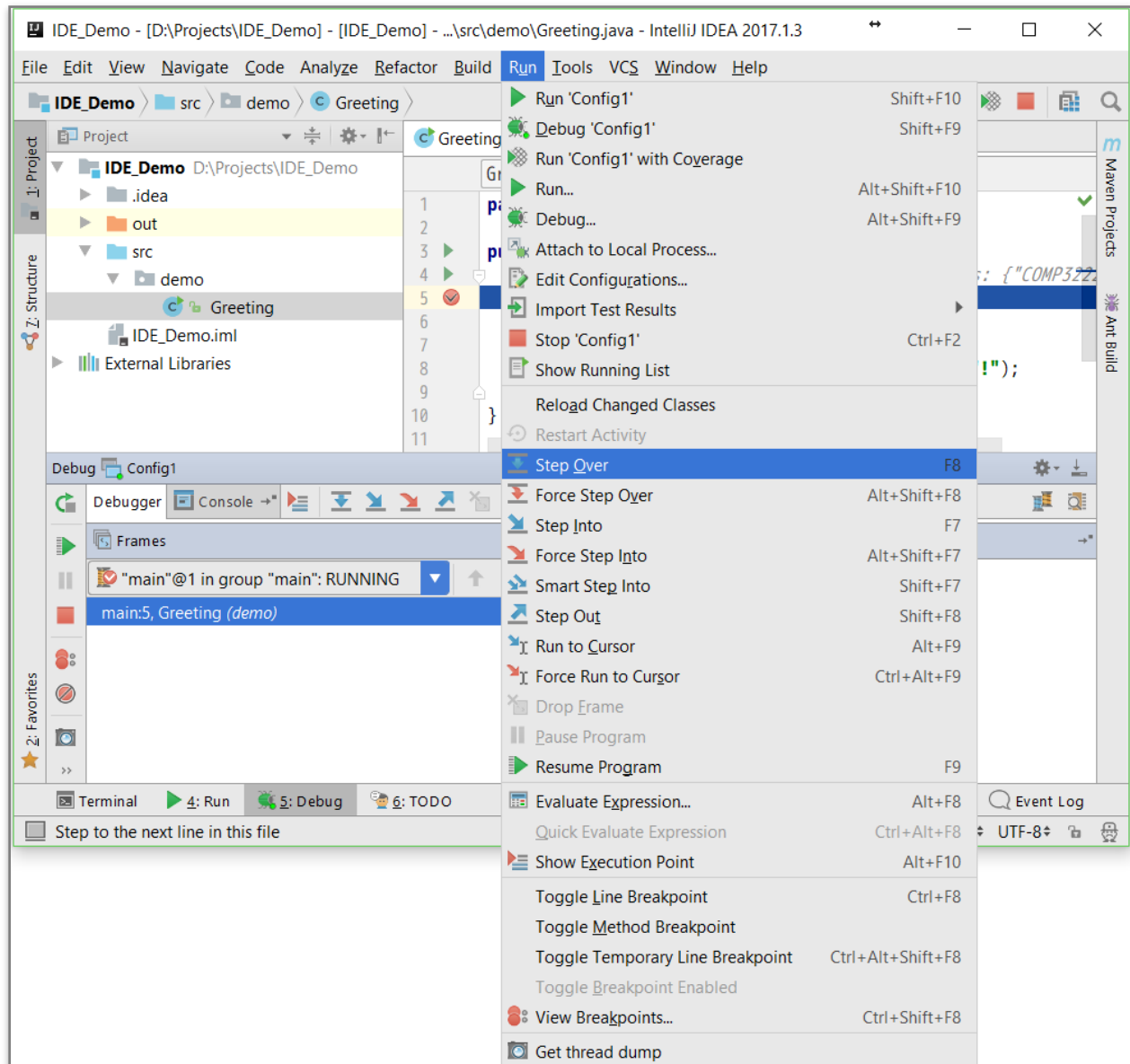
Start debugging by clicking on Run -> Debug 'Config1' or simply the Debug tool button.



During debugging, the program execution will stop before each line with an active breakpoint, allowing you to inspect the values of variables before executing that line.

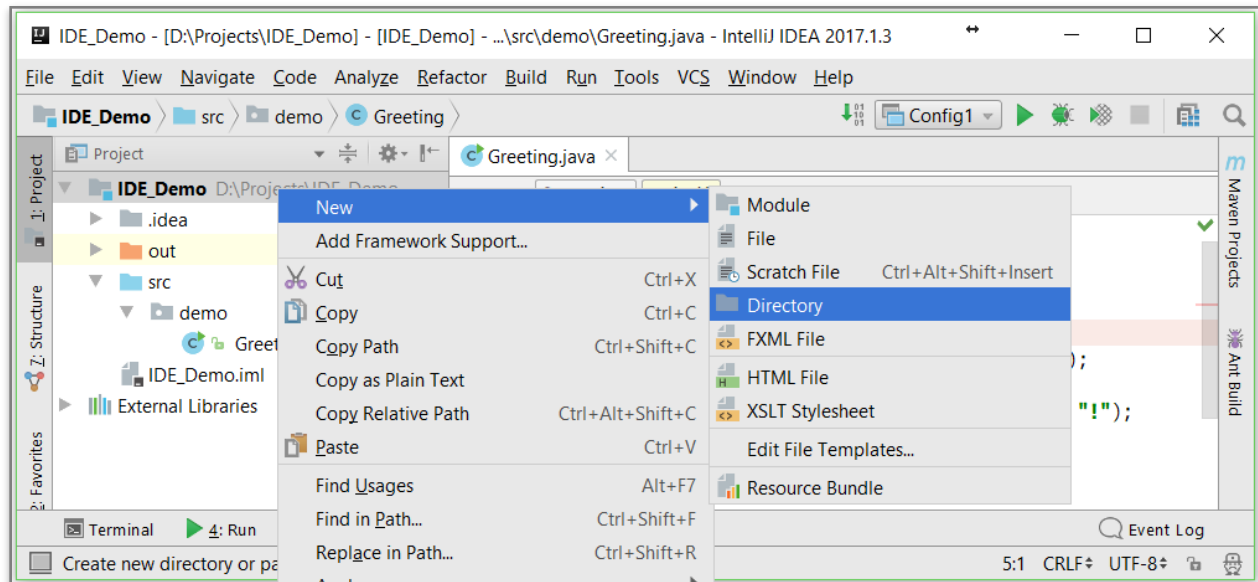


When the execution is stopped during debugging, you may also opt for executing the next statement in a single step (Step Over), executing the next statement in a single step unless it involves a method call, in which case the execution should follow the method call into the callee method (Step Into), or continuing the execution until the current method has returned to its caller (Step Out).

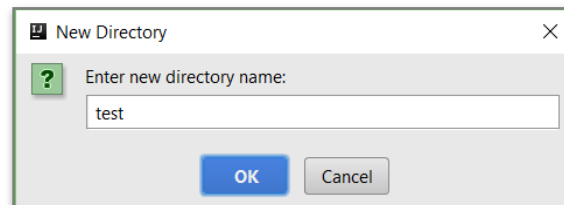


Adding Libraries

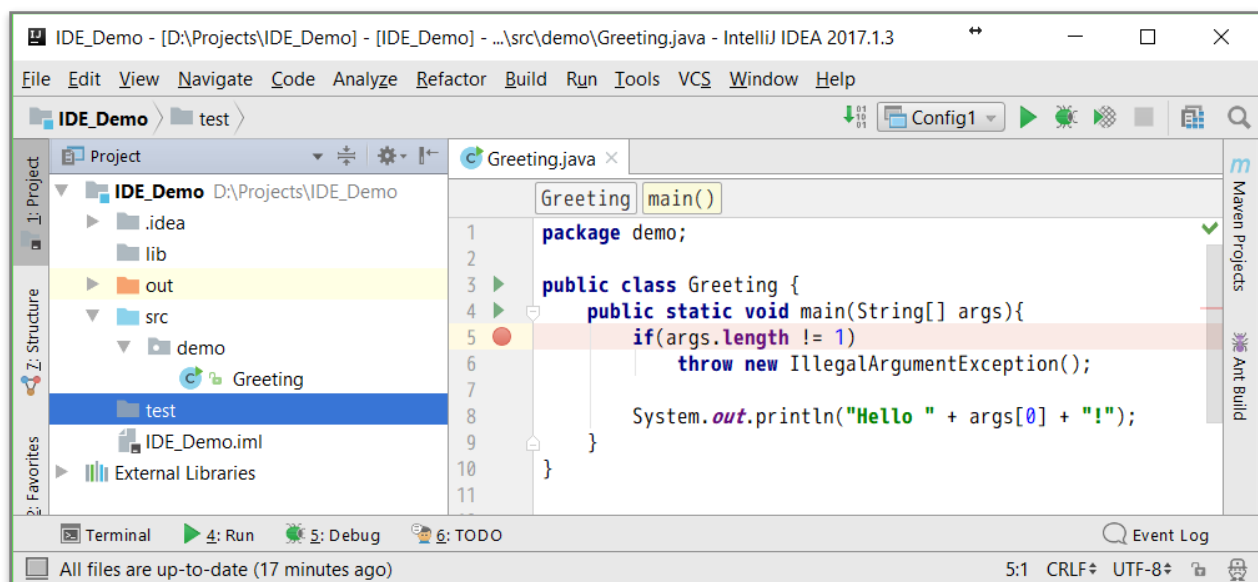
To add a new directory to the project, right click on the project node and select **New** -> **Directory**.



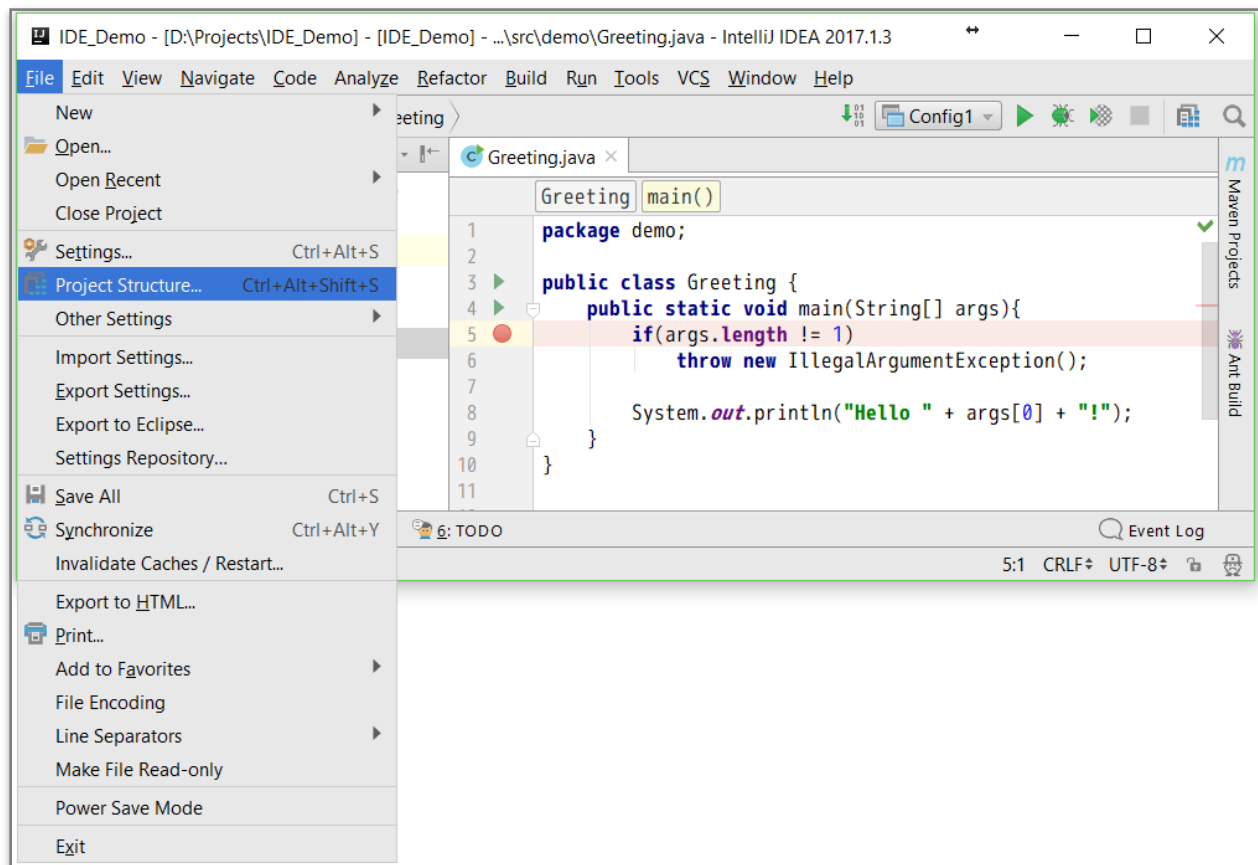
Name the project as you wish and then click on **OK**.



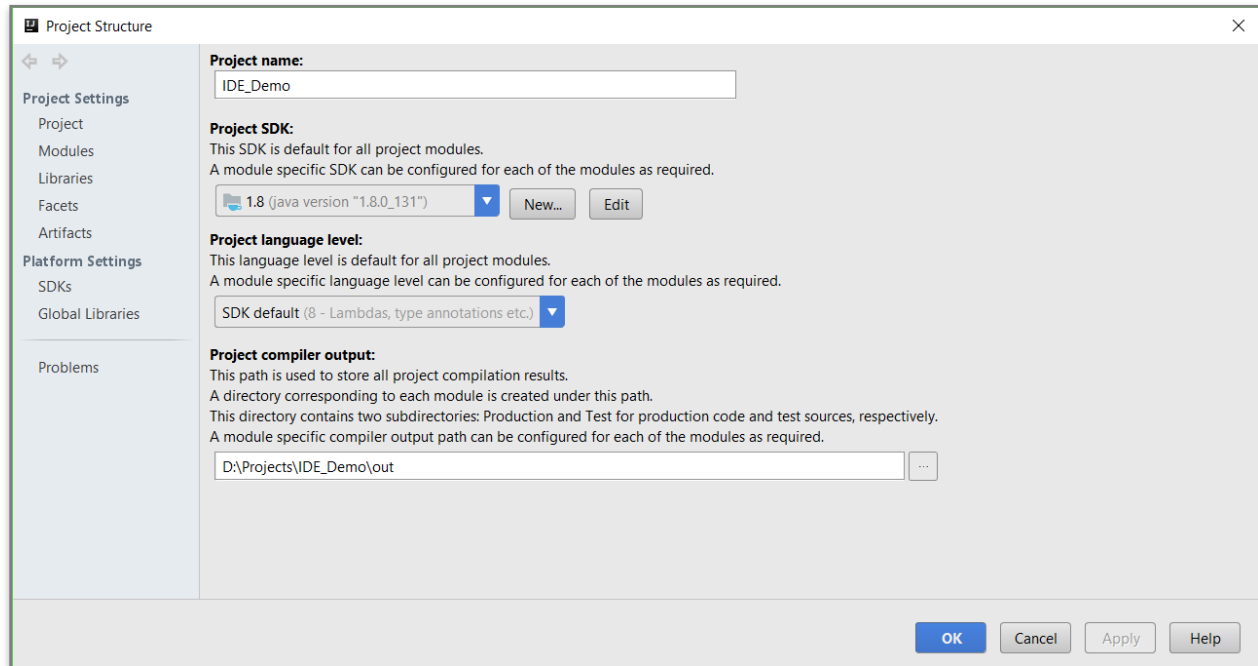
Add two directories **test** and **lib** to the project.



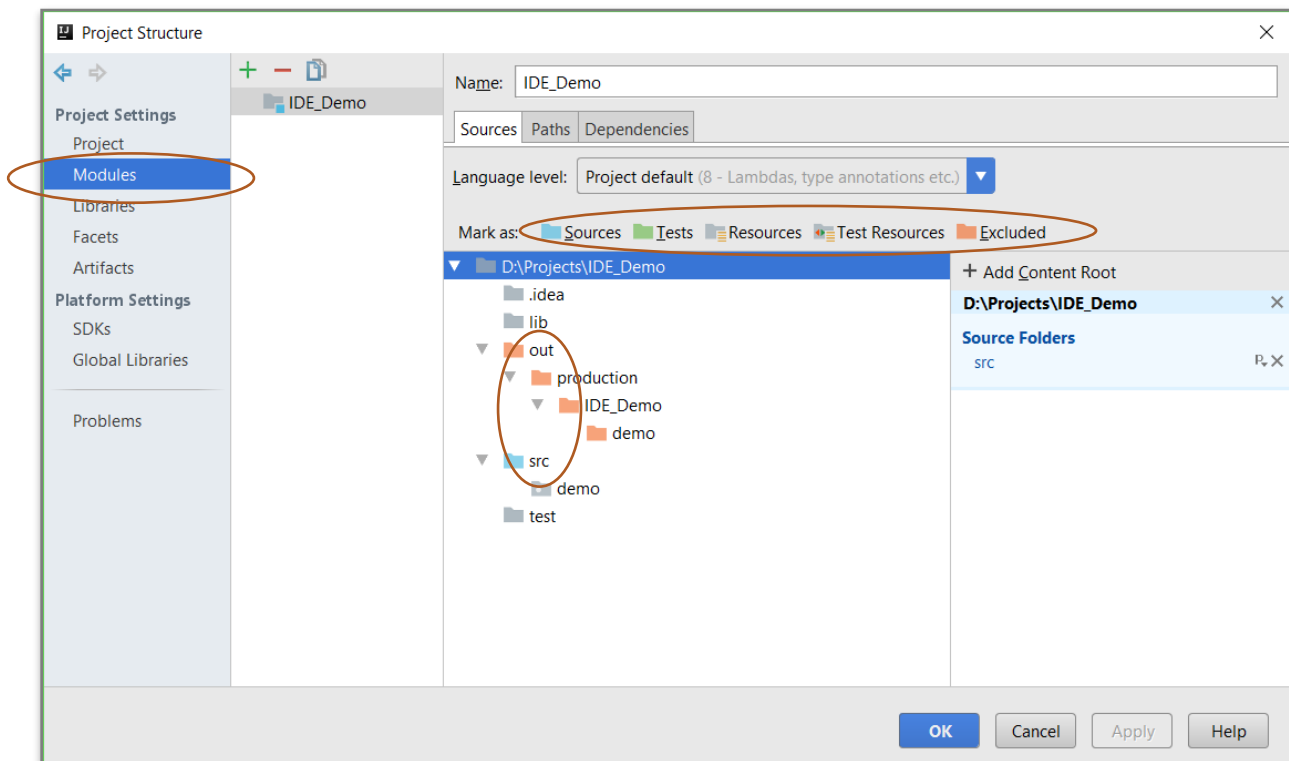
You can view the project structure by clicking on File -> Project Structure.



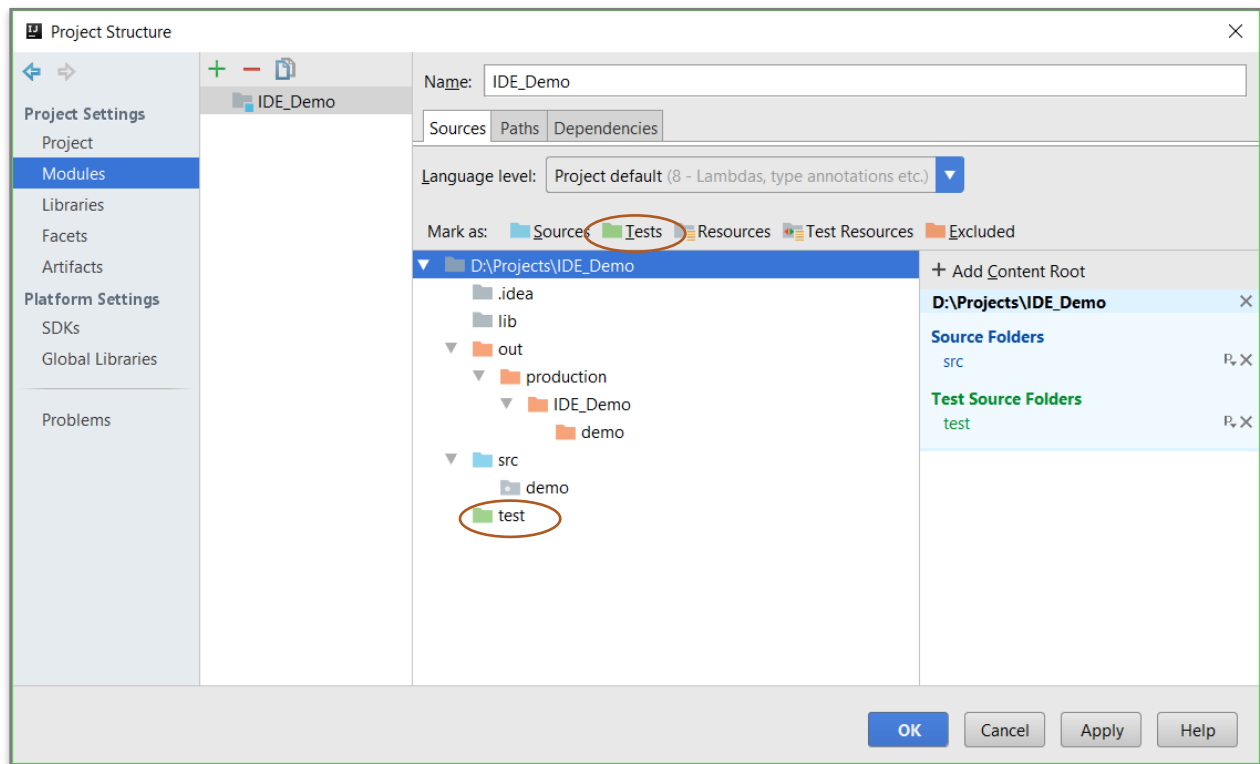
In the **Project Structure** dialog, you can view/change some basic information about the project. Note Java compiler will put the compilation results into folder **out**.



Select **Modules**, you should be able to see the directory structure of the project as the following. Note folder **src** is marked as **Sources** and folder **out** as **Excluded**.



Select folder `test` and mark it as `Tests`. Afterwards, the structure should look like the following. Click on `OK` to save the changes.

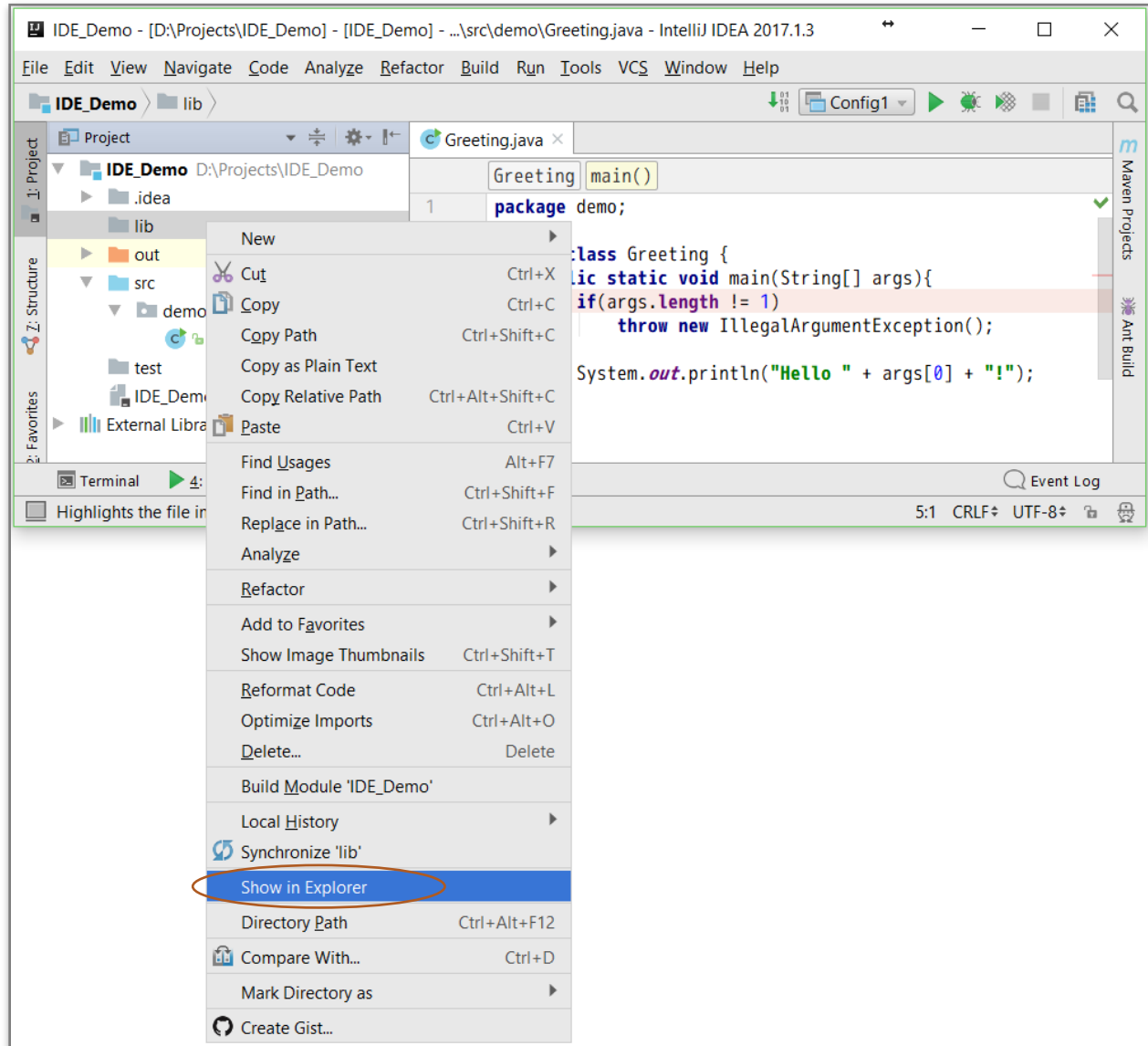


Right click on folder `lib` and then **Show in Explorer** to open the folder.

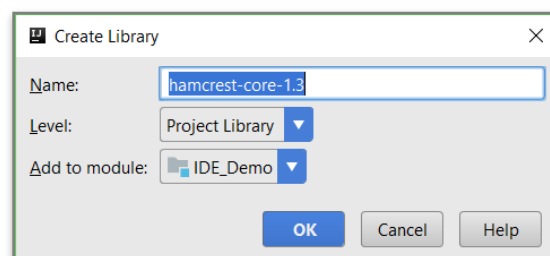
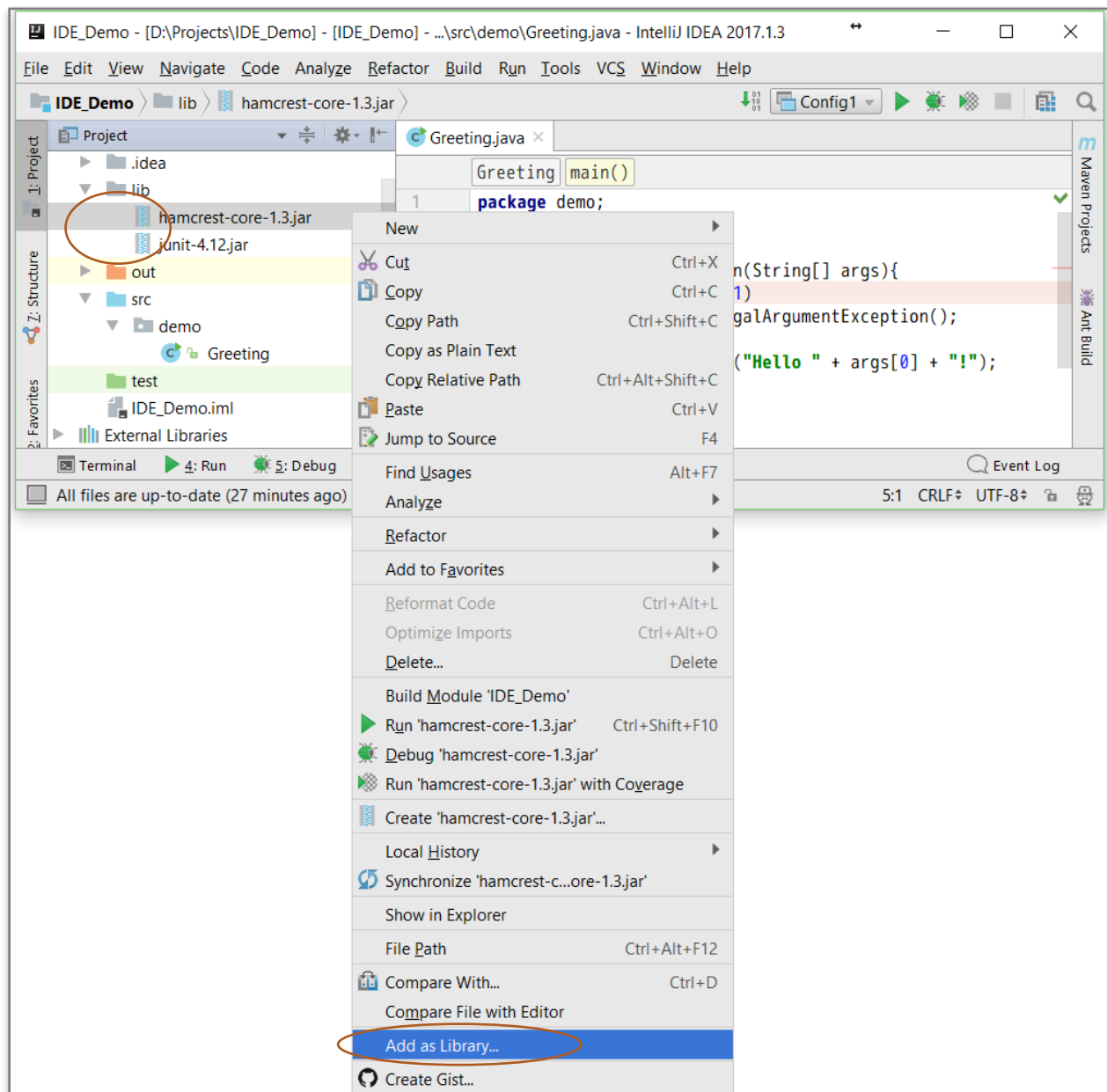
Copy the following two files from your IntelliJ IDEA installation to the `lib` folder. Note the exact path and file name may be slightly different depending on your system.

C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2017.1.3\lib\hamcrest-core-1.3.jar

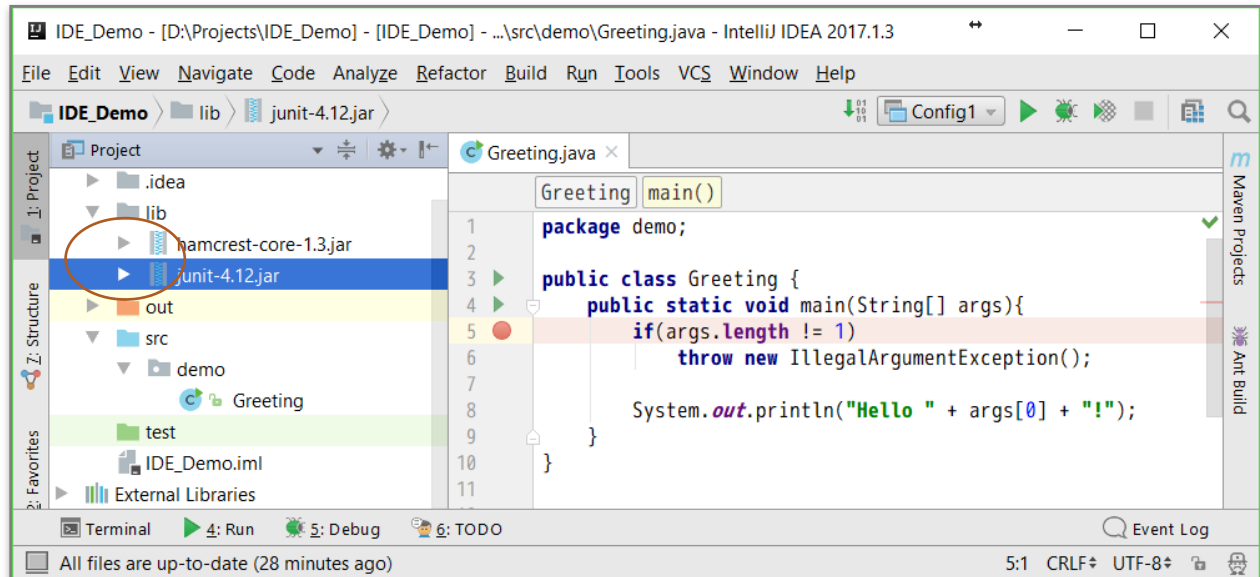
C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2017.1.3\lib\junit-4.12.jar



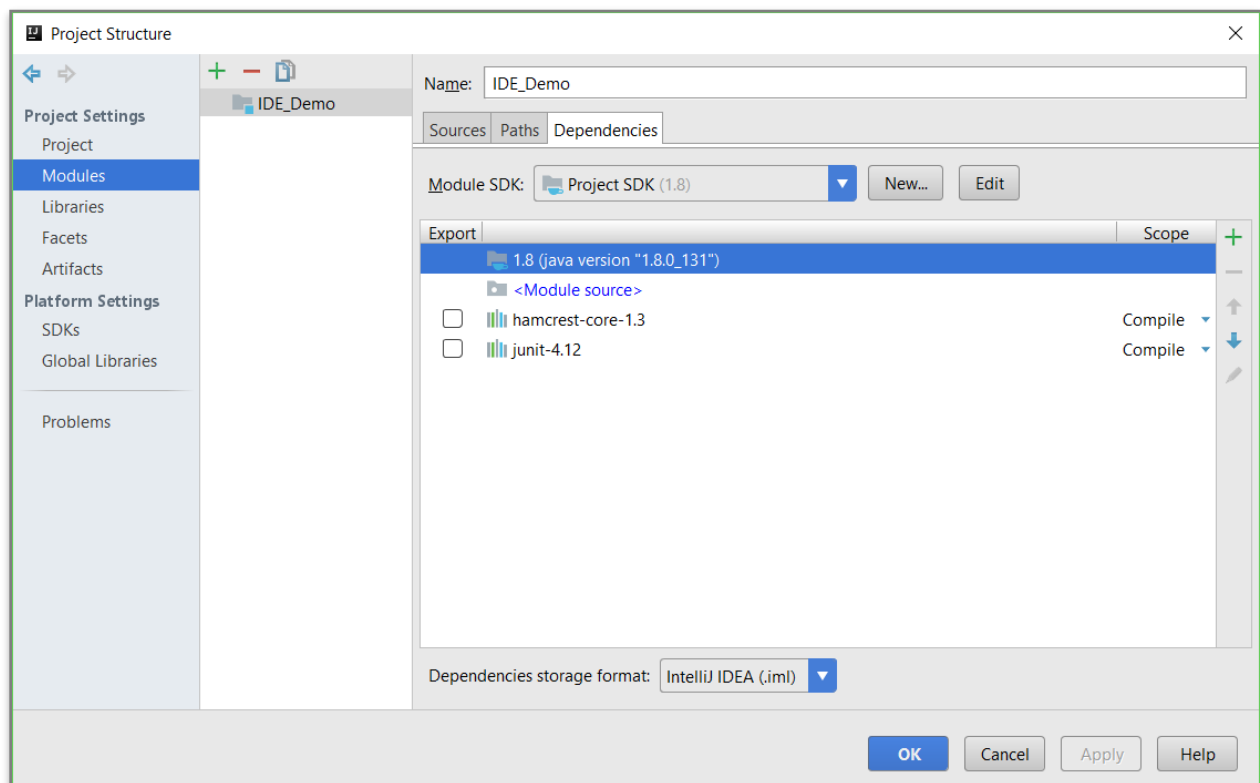
The two jar files should appear in folder `lib` in your IDE now, but they are not part of the project yet. Right click on a jar file and select **Add as Library...** and then click on **OK** in the pop up dialog. Do the same with the other jar file.



Now the two Jar files are added to the project as libraries. These two Jar files provides the JUnit testing framework.



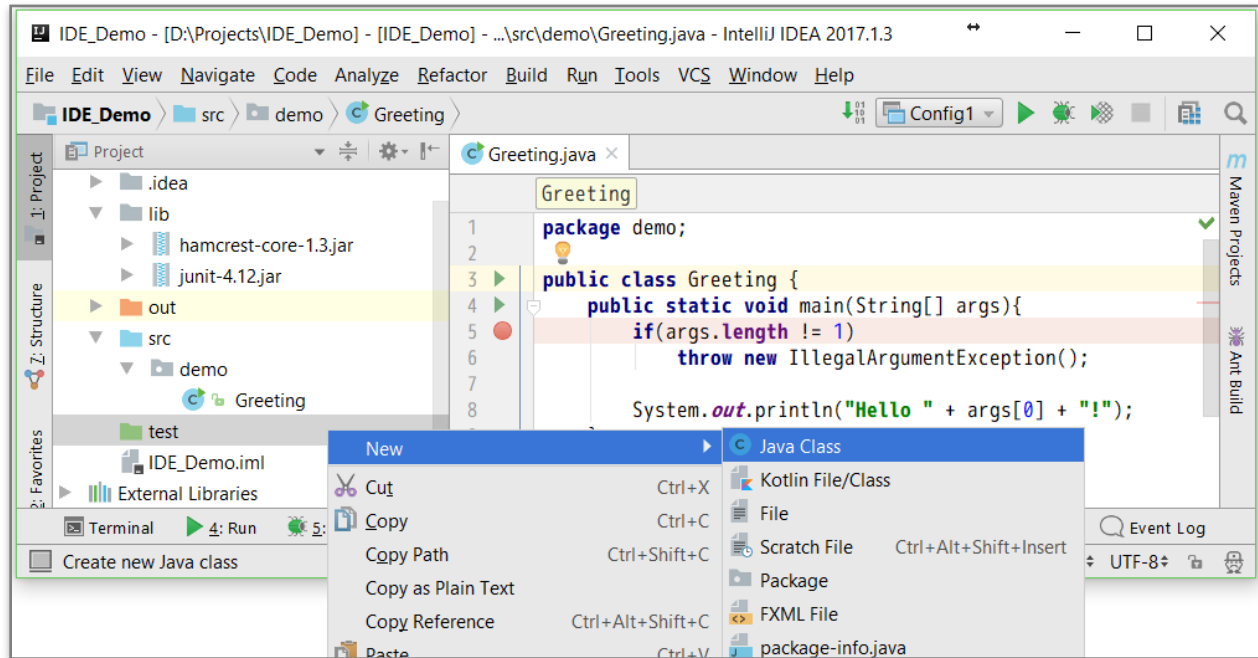
If you open Project Structure again, you will see the two Jar files have indeed been added as libraries.



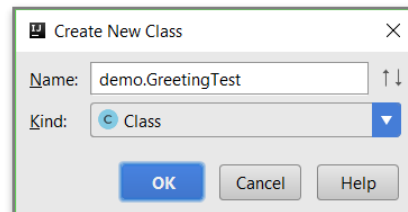
Testing Your Code

Detailed instructions at IntelliJ IDEA Help: <https://www.jetbrains.com/help/idea/create-tests.html>

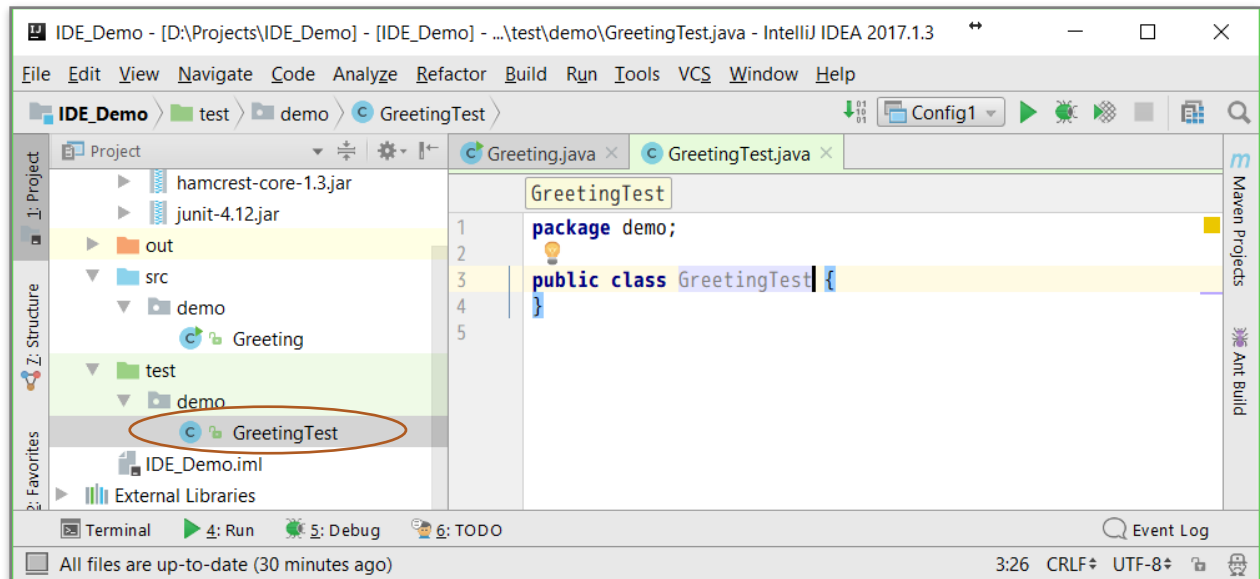
Next, we create a JUnit test class by right clicking on the test folder and then selecting **New** -> **Java Class**.



Provide the fully qualified name for your test class, then click on **OK**.

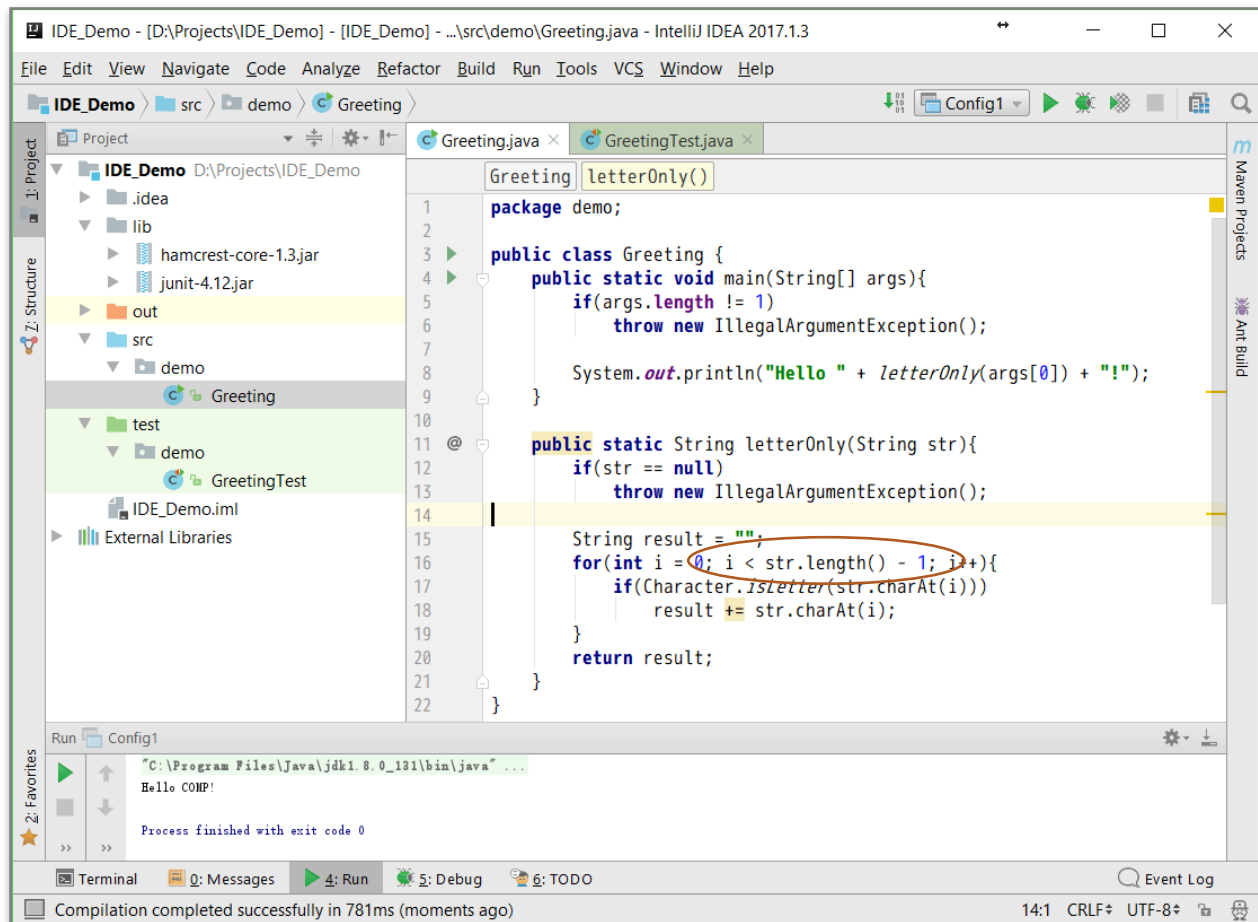


Double click on the test class name to open its source in Editor.

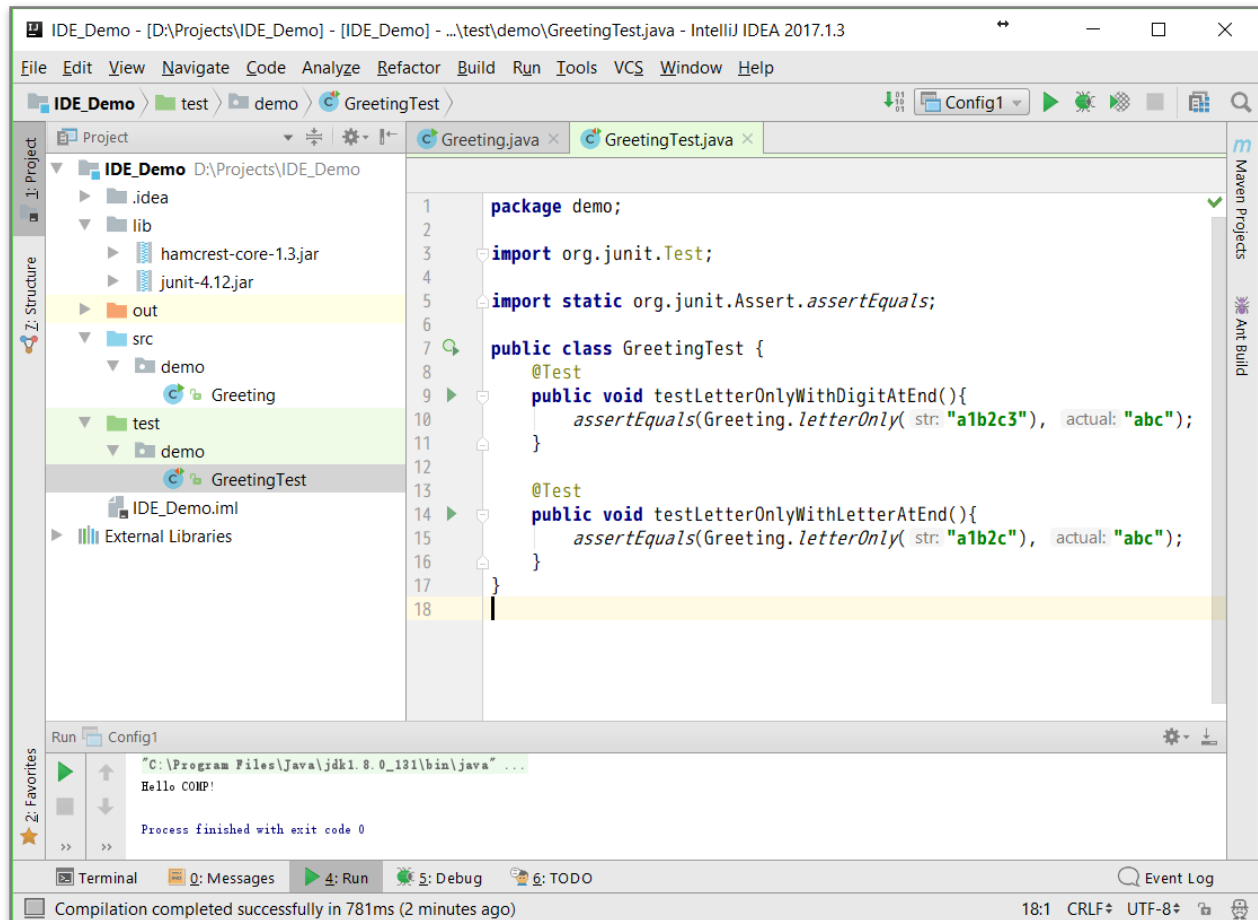


Before writing any test method, we first add a new `public static` method to class `Greeting`. The method takes a `String` and returns a new `String` with only letters from the argument left. The `main` method of the application invokes the new method to process its argument before printing it out.

Note the loop-continuation condition in the implementation is wrong, and the correct condition should be `i < str.length()`. The mistake will only cause a problem if the argument ends with a letter.



Next, we write two tests for the new method, one using a string that ends with a digit and the other a string that ends with a letter. Annotation `@Test` turns a `public void` argument-less method into a test method, while `assertEquals` will use `equals()` method to compare its two arguments and throw an exception if they are not equal.



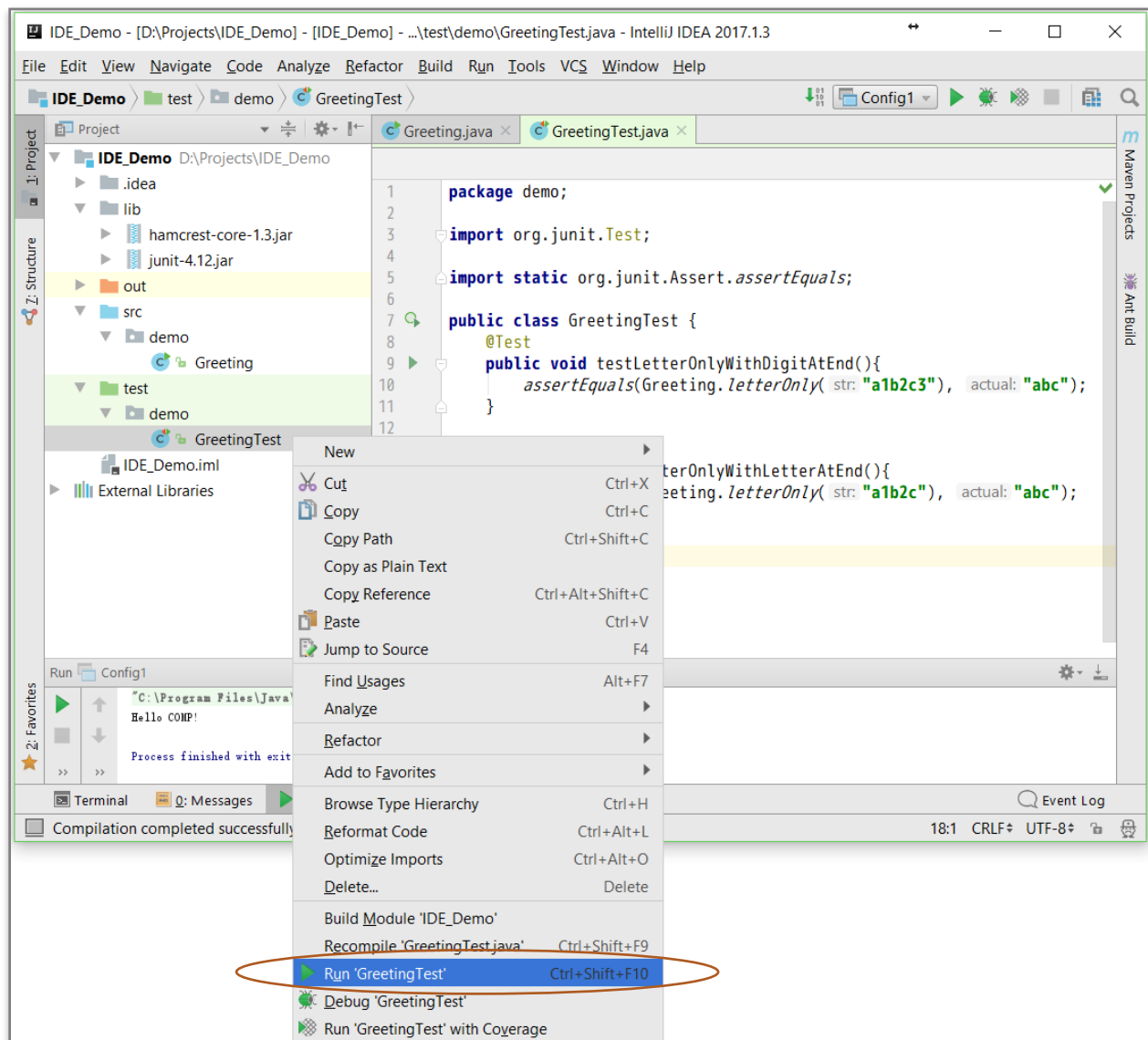
Tests that terminate without exceptions are in general considered successful, but JUnit does provide a way to specify that a test is expected to terminate with an exception of specific type, as shown in the example below.

```

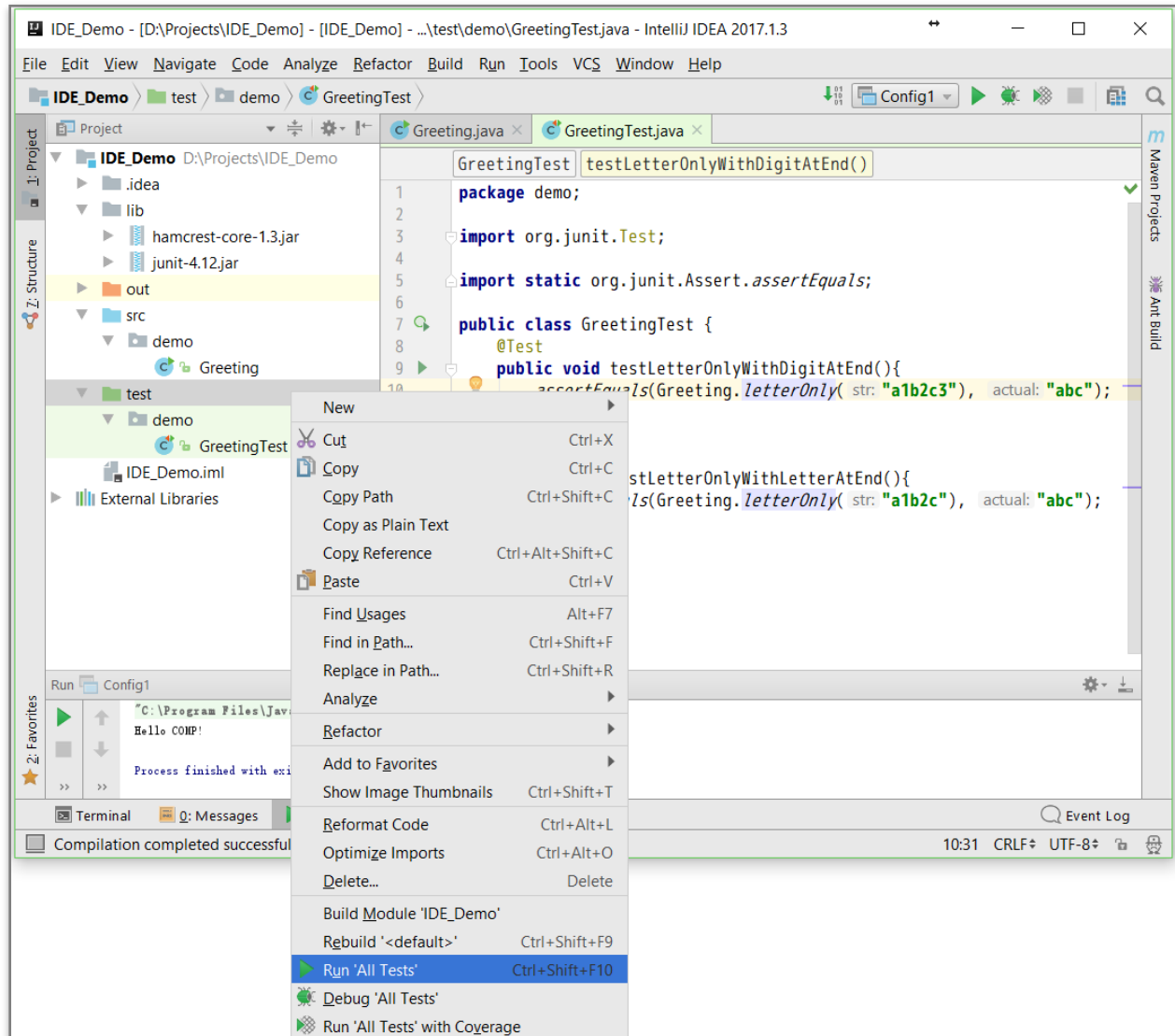
@Test(expected=NullPointerException.class)
public void testNullPointerException(){
    String s = null;
    System.out.println(s.toString());
}

```

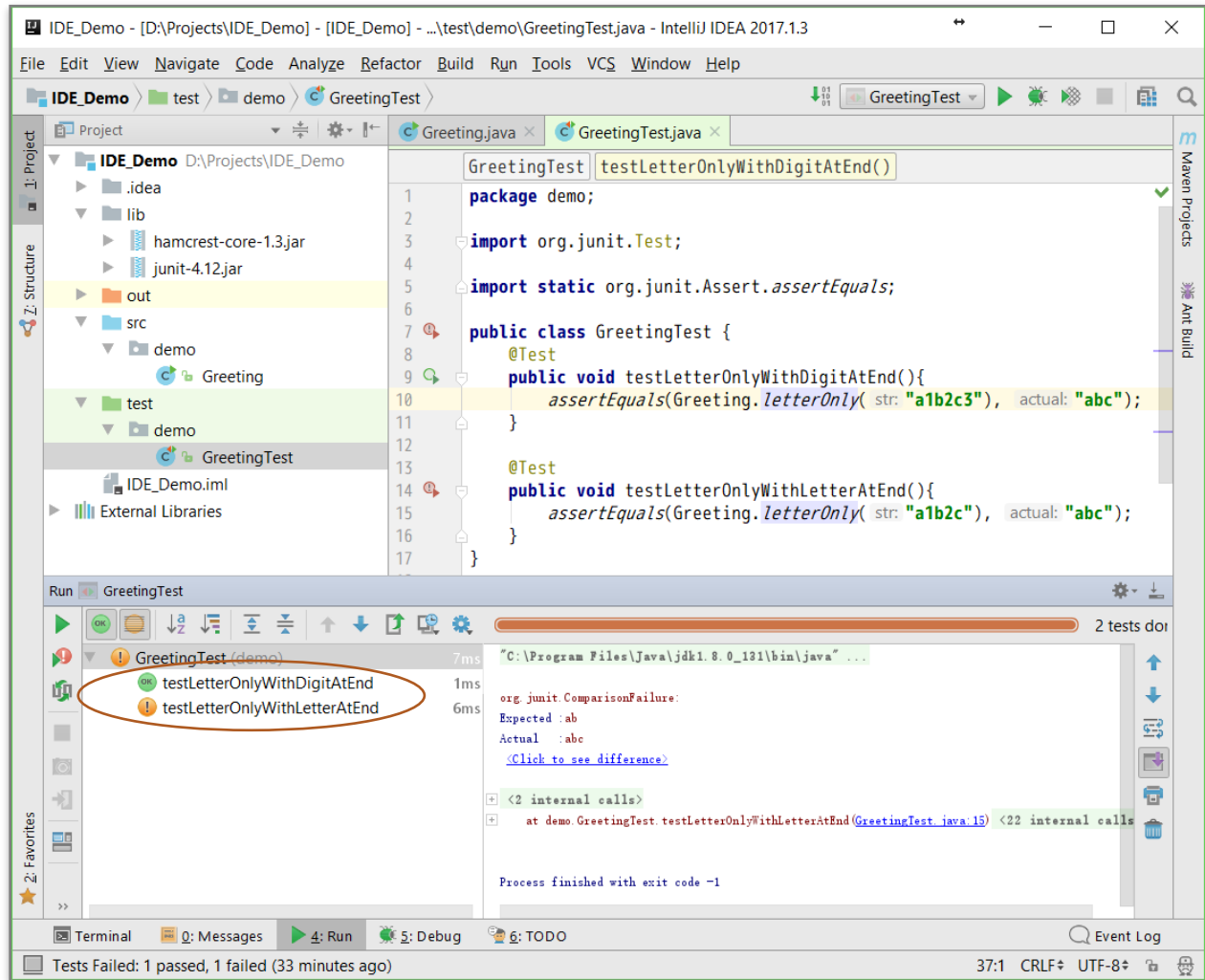
Once we have finished the test class, we can right click on the test class and select Run 'GreetingTest' to run the tests. If you prefer, you can also right click on a test method and run only that single test.



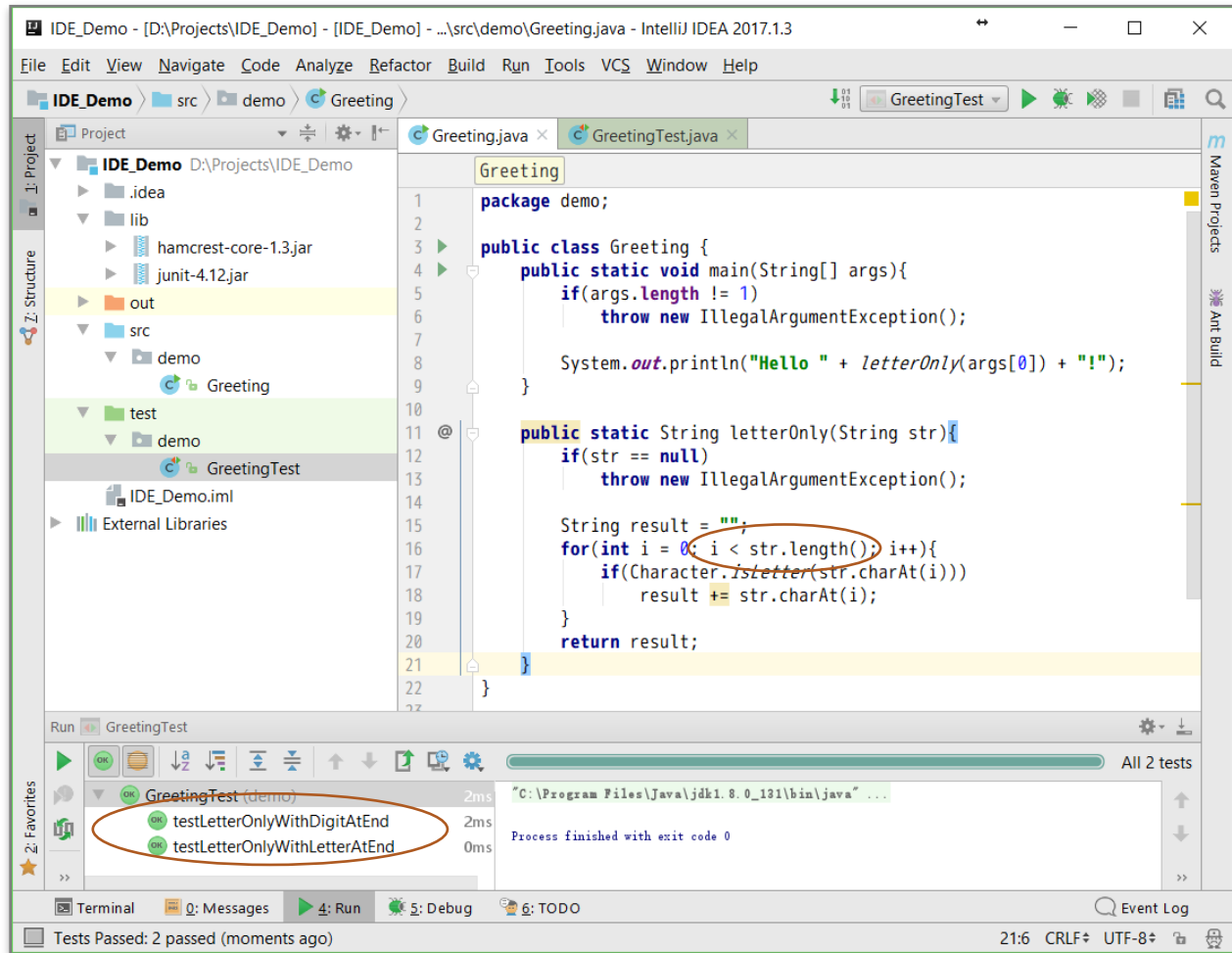
In case you have multiple test classes and want to run all of them, you can right click on folder test and select Run 'All Tests'.



JUnit will run the tests automatically and report the results. You can easily see which tests have passed and which ones have failed. Note that each test method is a test.



If we correct the mistake in method `letterOnly` and run the tests again, we will see all the test become passing.



To measure the code coverage of the tests, we run the tests with Coverage.



When testing is finished, coverage information will be shown in the Coverage tool window.

The screenshot shows the IntelliJ IDEA 2017.1.3 interface. The main editor displays the `Greeting.java` file with the following code:

```
1 package demo;
2
3 public class Greeting {
4     public static void main(String[] args) {
5         if(args.length != 1)
6             throw new IllegalArgumentException("Invalid argument");
7         System.out.println("Hello, " + args[0]);
8     }
9 }
10
11 @Test
12 public static String letterOnly(String str) {
13     if(str == null)
14         throw new IllegalArgumentException("Invalid argument");
15     String result = "";
16     for(int i = 0; i < str.length(); i++) {
17         if(Character.isLetter(str.charAt(i))) {
18             result += str.charAt(i);
19         }
20     }
21     return result;
22 }
```

The Coverage tool window is open, showing the following table:

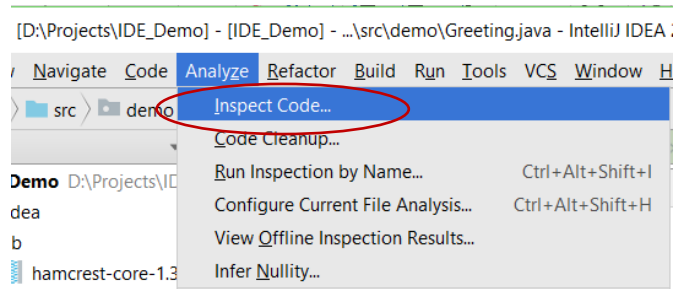
| Element | Class, % | Method, % | Line, % |
|---------|------------|-----------|------------|
| demo | 100% (1/1) | 50% (1/2) | 50% (6/12) |

The table indicates that 100% of classes and 50% of lines are covered in the 'demo' package. The Run tool window shows the execution of the `GreetingTest` test, with the following output:

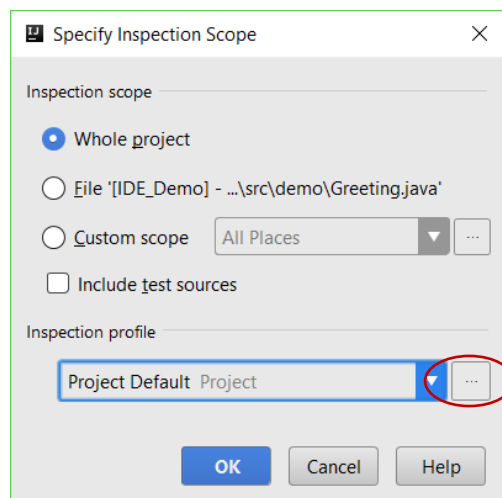
```
IntelliJ IDEA coverage runner
sampling ...
include patterns:
demo\.*
exclude patterns:
Process finished with exit code 0
```

Running The Inspection Tool

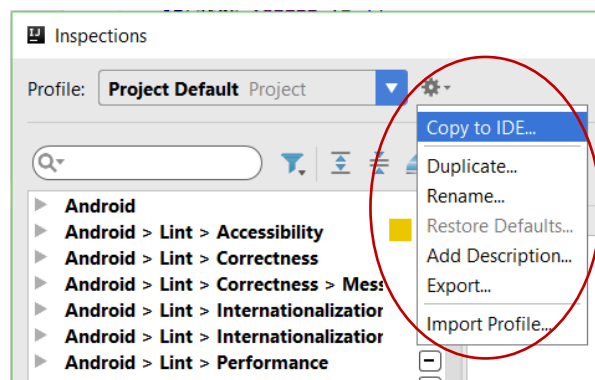
Go to **Analyze -> Inspect Code...**



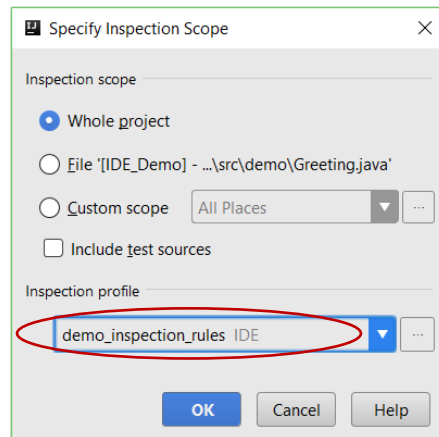
In the dialog, select the **Inspection scope** and uncheck **Include test sources**. Then click on the ellipses at the bottom-right corner of the dialog.



Click on the gear button, then **Import Profile...**



Import the desired inspection rules from an XML file and click on **OK** to go back to the previous dialog. Select the imported profile as the active **Inspection profile**.



Click on **OK** to run the inspection. Inspection results will be shown in the **Inspection Results** tool panel.

