



[Testing](#) / [Testing frameworks](#) / [JUnit](#) / [Get started with JUnit](#)

JUnit 5

Last modified: 27 December 2023

In this tutorial, you will learn how to set up JUnit for your projects, create tests, and run them to see if your code is operating correctly. It contains just the basic steps to get you started.

If you want to know more about JUnit, refer to [the official documentation ↗](#). To learn more about testing features of IntelliJ IDEA, refer to other topics in this section.

You can choose to follow the tutorial using either Maven, Gradle, or the IntelliJ builder.

Maven

[Gradle](#)

IntelliJ build tool

Create a project

1. In the main menu, go to **File | New | Project**.
2. Select **New Project**. Specify the name for the project, for example, `junit-tutorial`.
3. Select **Gradle** as a build tool. In **Language**, select **Java**.
4. From the **JDK** list, select the **JDK** that you want to use in your project.

If the JDK is installed on your computer, but not defined in the IDE, select **Add JDK** and specify the path to the JDK home directory.

If you don't have the necessary JDK on your computer, select **Download JDK**.

5. Click **Create**.



For more information about working with Gradle projects, refer to [Gradle](#).

Add dependencies

For our project to use JUnit features, we need to add JUnit as a dependency.

1. Open **build.gradle** in the root directory of your project.




To quickly navigate to a file, press **Ctrl** **Shift** **N** and enter its name.

2. In **build.gradle**, press **Alt** **Insert** and select **Add Maven artifact dependency**.

3. In the tool window that opens, type `org.junit.jupiter:junit-jupiter` in the search field.

Locate the necessary dependency in the search results and click **Add**.

4. When the dependency is added to **build.gradle**, press **Ctrl** **Shift** **O** or click  in the **Gradle** tool window to import the changes.



The procedure above shows the 'manual' way so that you know what happens behind the scenes and where you set up the testing framework. However, if you just start writing tests, IntelliJ IDEA will automatically detect if the dependency is missing and prompt you to add it.

Write application code

Let's add some code that we'll be testing.

1. In the **Project** tool window **Alt** **1**, go to `src/main/java` and create a Java file called `Calculator.java`.

2. Paste the following code in the file:

```
import java.util.stream.DoubleStream;

public class Calculator {

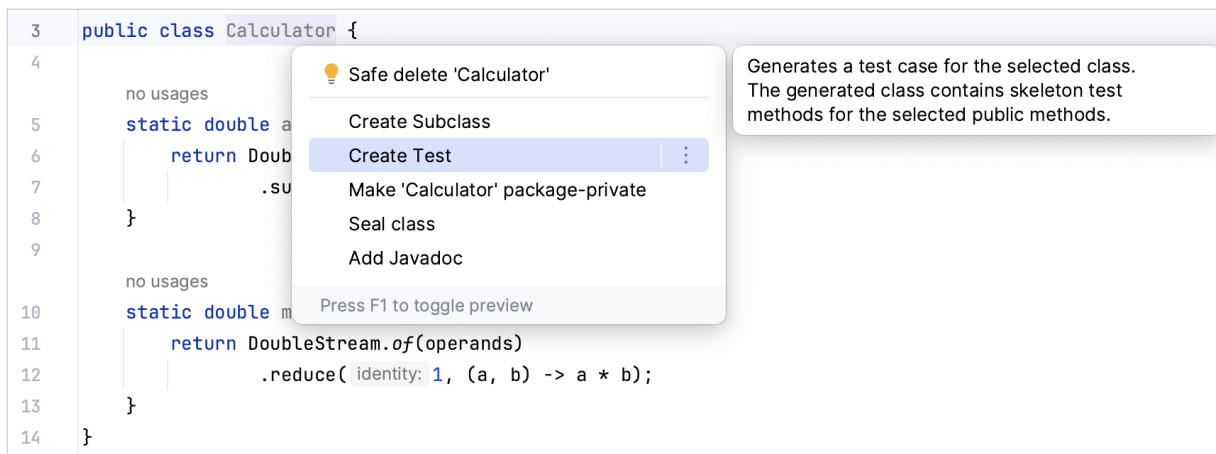
    static double add(double... operands) {
        return DoubleStream.of(operands)
            .sum();
    }

    static double multiply(double... operands) {
        return DoubleStream.of(operands)
            .reduce(1, (a, b) -> a * b);
    }
}
```

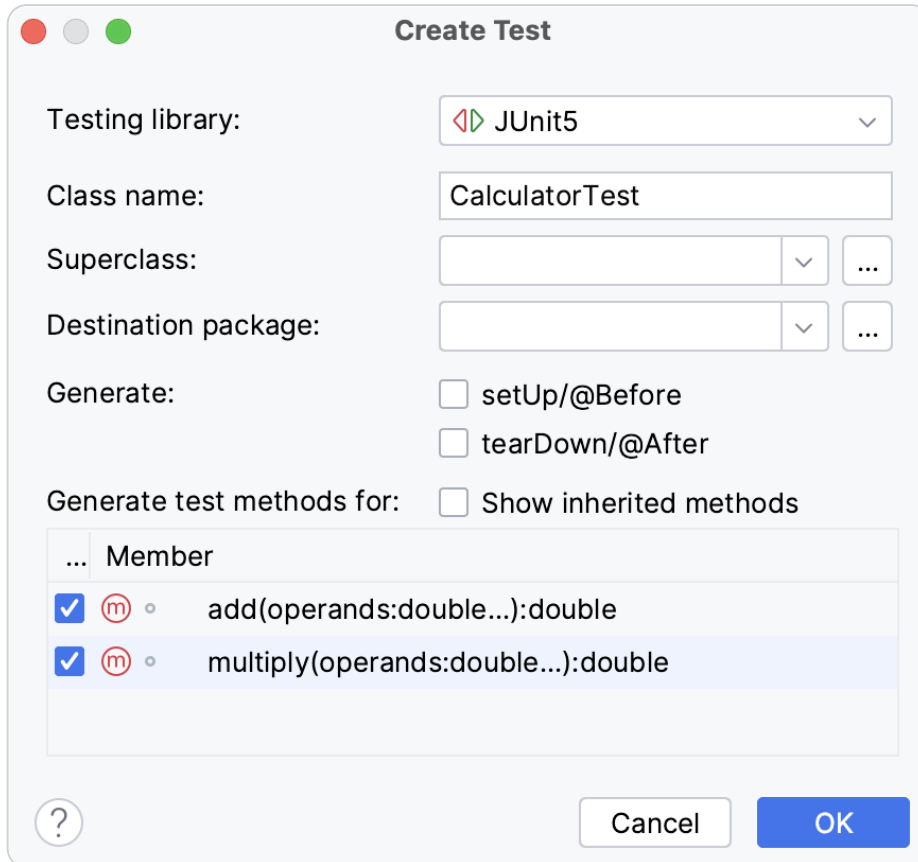
Create tests

Now let's create a test. A **test** is a piece of code whose function is to check if another piece of code is operating correctly. In order to do the check, it calls the tested method and compares the result with the predefined **expected result**. An expected result can be, for example, a specific return value or an exception.

1. Place the caret at the `Calculator` class declaration and press `Alt Enter`. Alternatively, right-click it and select **Show Context Actions**. From the menu, select **Create test**.



2. Select the two class methods that we are going to test.



Create Test

Testing library: JUnit5

Class name: CalculatorTest

Superclass:

Destination package:

Generate: ☐ setUp/@Before ☐ tearDown/@After

Generate test methods for: ☐ Show inherited methods

...	Member
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> add(operands:double...):double
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> multiply(operands:double...):double

Cancel OK

3. The editor takes you to the newly created test class. Modify the `add()` test as follows:

```
@Test
@DisplayName("Add two numbers")
void add() {
    assertEquals(4, Calculator.add(2, 2));
}
```

This simple test will check if our method correctly adds 2 and 2. The `@DisplayName` annotation specifies a more convenient and informative name for the test.

4. Now what if you want to add multiple assertions in a single test and execute all of them regardless of whether some of them fail? Let's do it for the `multiply()` method:

```
@Test
@DisplayName("Multiply two numbers")
```

```
void multiply() {
    assertAll(() -> assertEquals(4, Calculator.multiply(2, 2)),
        () -> assertEquals(-4, Calculator.multiply(2, -2)),
        () -> assertEquals(4, Calculator.multiply(-2, -2)),
        () -> assertEquals(0, Calculator.multiply(1, 0)));
}
```

The `assertAll()` method takes a series of assertions in form of lambda expressions and ensures all of them are checked. This is more convenient than having multiple single assertions because you will always see a granular result rather than the result of the entire test.

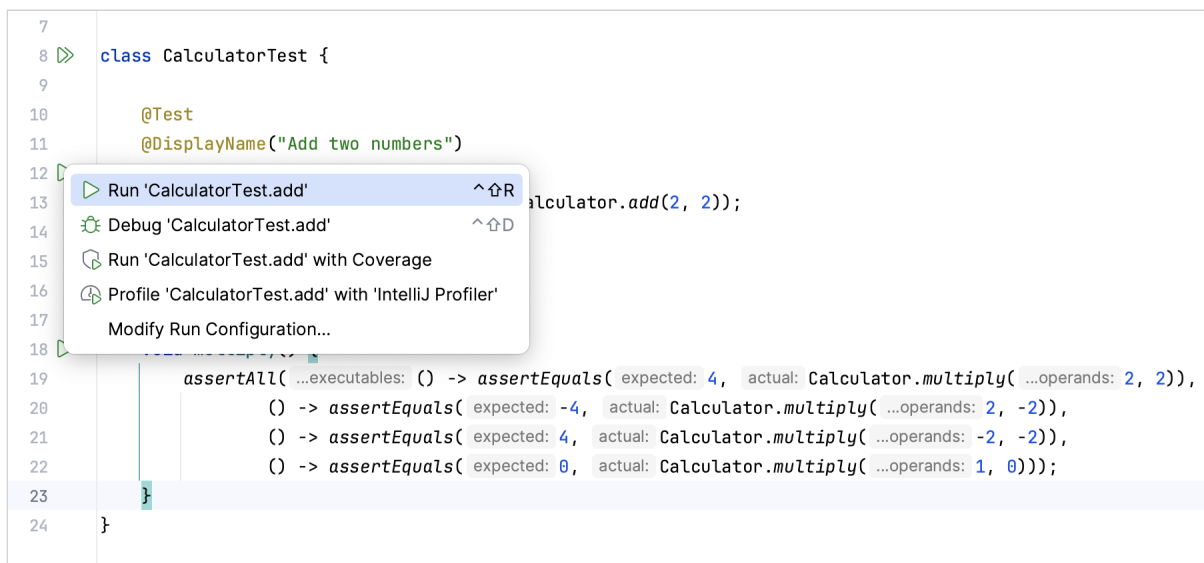


To navigate between the test and the code being tested, use the **Ctrl** **Shift** **T** shortcut.

Run tests and view their results

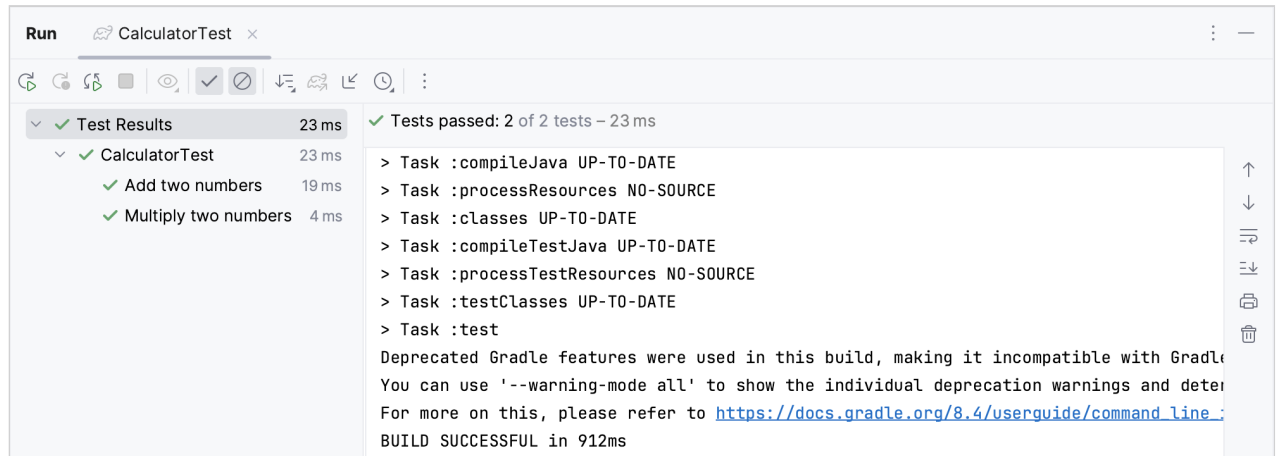
After we have set up the code for the testing, we can run the tests and find out if the tested methods are working correctly.

- To run an individual test, click  in the gutter and select **Run**.



- To run all tests in a test class, click  against the test class declaration and select **Run**.

You can view test results in the **Run** tool window.



IntelliJ IDEA hides passed tests by default. To see them, make sure the **Show Passed** option is enabled in the **Run** tool window.