**Hands On Lab 1  
[Test Cases and Test Fixtures](https://personales.unican.es/corcuerp/java/Labs/LAB_23.htm" \l "Exercise_1)**

### IntelliJ

Many prefer to execute their tests from the convenience of an automated integrated development environment such as IntelliJ.

### Write application code﻿

To see IntelliJ’s JUnit capabilities in action, create a new Java project and add the Philadelphia class.

**public** **class** Philadelphia {

**public** **static** **boolean** isItSunny() {

**return** **true**;

}}

### Add dependencies﻿

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

Open pom.xml in the root directory of your project.

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

In pom.xml, press Alt + Insert and select Dependency.

In the dialog that opens, type **org.junit.jupiter:junit-jupiter** in the search field.

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

When the dependency is added to pom.xml, press Ctrl+Shift+O or click in the Maven tool window to import the changes.

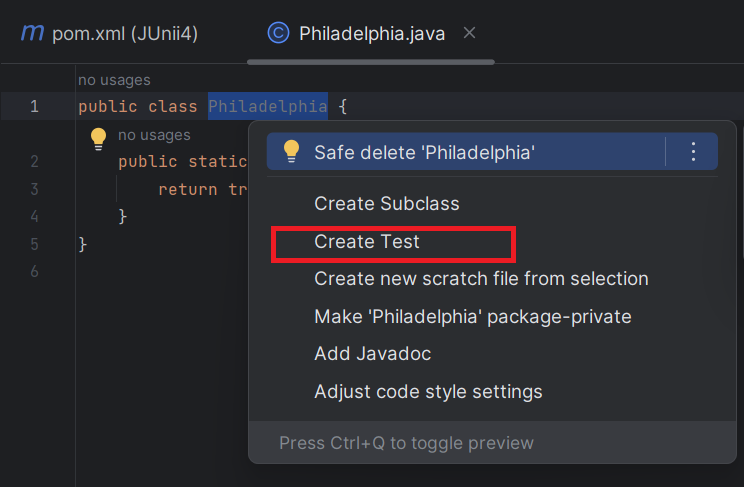


The procedure above shows the 'manual' way so that you know what happens behind the scenes and where you set up the testing framework. However, if you just start writing tests, IntelliJ IDEA [will automatically detect if the dependency is missing and prompt you to add it](https://www.jetbrains.com/help/idea/testing.html#add-testing-libraries).

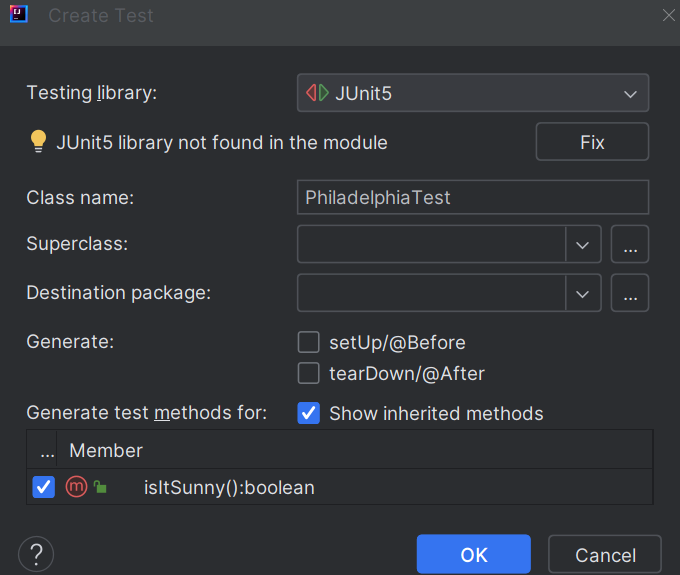
### Create tests﻿

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.

Place the caret at the Philadelphia class declaration and press Alt + Enter. Alternatively, right-click it and select Show Context Actions. From the menu, select Create Test.

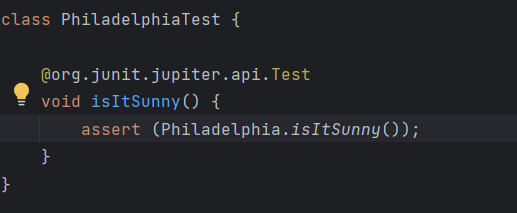


The editor takes you to the newly created test class.



We'll be testing the "isItSunny" method, so select the checkbox next to that method as seen above:

Modify the isItSunny () test as follows:

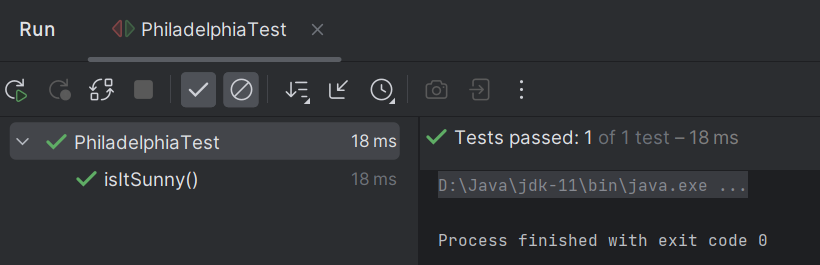


### Run tests and view their results﻿

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.

**Exercise**Perform the following tasks to become familiar with jUnit basics (you may use either the command line or IntelliJ for this exercise):

* Run the test described above and verify that you get the same output.
* Add an additional test which fails an assertion (this can be as simple as a test containing only an "assertTrue(false);" statement) and run the tests again.  Describe what has changed in the test results.
* Add an additional test that throws an exception and run the tests again.  Describe what has changed in the test results.

## Test Fixtures

Often when testing you'll need to do the same basic set-up or tear-down activities for several tests.  jUnit provides an easy way to aggregate tests and handle the set-up and tear-down activities in separate methods that get called before and after running each test.  To use this capability, you need only group such tests into the same containing class and annotate the set-up and tear-down methods with "org.junit.Before" and "org.junit.After" respectively.  For example, the following class represents a very simple test fixture that exercises some of the methods found in the "java.util.ArrayList" class:

import **static** org.junit.Assert.\*;

import java.util.ArrayList;import java.util.Arrays;

import java.util.List;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

**public** **class** ArrayListTest {

List<Integer> testArray;

/\*\* \* This method is invoked before each test is run to set up the test array \* with a known set of values. \*/

@Before

// Informs JUnit that this method should be run before each test

**public** **void** setUp() {

testArray = **new** ArrayList<Integer>(Arrays.asList(3, 1, 4, 1, 5));

}

/\*\* \* This method is invoked after each test is run to perform tear down \* activities (not needed in this test fixture). \*/

@After

// Informs JUnit that this method should be run after each test

**public** **void** tearDown() {

// No tear down needed for this test

}

/\*\* \* Adds a value to the array and verifies the add was successful. \*/

@Test

**public** **void** testAdd() {

testArray.add(9);

List<Integer> expected =

**new** ArrayList<Integer>(Arrays.asList(3, 1, 4, 1, 5, 9));

assertEquals(testArray, expected);

}

/\*\* \* Removes a value from the array and verifies the remove was successful. \*/

@Test

**public** **void** testRemoveObject() {

testArray.remove(**new** Integer(5));

List<Integer> expected =

**new** ArrayList<Integer>(Arrays.asList(3, 1, 4, 1));

assertEquals(testArray, expected);

}

/\*\* \* Tests the indexOf method and verifies the expected return value. \*/

@Test

**public** **void** testIndexOf() {

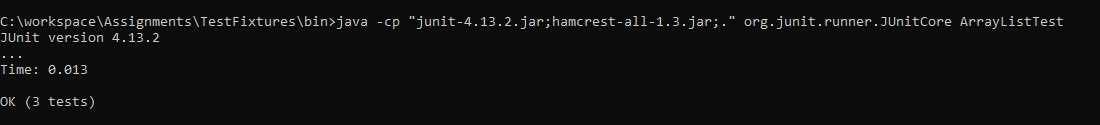
assertEquals(testArray.indexOf(4), 2);

}

}

Each of the tests above works on the same pre-initialized test array named "testArray".  Also, it's important to note that each jUnit test is run independently: actions taken in one test are not seen in any other test.  In other words, the set-up and tear-down are run for each test. Run the tests in this class from the command line. Junit4.jar should be in the same directory or the classpath.

Upon running the tests in this class, we'll get the following output:



**Exercise**  
Perform the following tasks to become familiar with jUnit basics:

* Run the test fixture described above and verify that you get the same output.
* Add an additional test which uses the testArray, tests the "clear" method, and verifies that the array is empty.
* Add an additional test which uses the testArray and tests the "contains" method by verifying it returns true when supplied a value that exists in the array.
* Add an additional test which uses the testArray and tests the "contains" method by verifying it returns false when supplied a value that does not exist in the array.
* Add an additional test which uses the testArray and tests the "get" method verifying it returns the correct value for a given index.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*