This tutorial was adopted from <https://www.tutorialspoint.com/junit/index.htm> but modified to work with Maven, utilize Eclipse and be contained within a single project. It also removes the redundancy from the tutorial.

What is JUnit?

JUnit is a unit-testing framework for Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks collectively known as xUnit, that originated with JUnit.

Testing is the process of checking the functionality of an application to ensure it runs as per requirements. Unit testing comes into picture at the developers’ level; it is the testing of single entity (class or method).

JUnit promotes the idea of "first testing then coding", which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented. This approach is like "test a little, code a little, test a little, code a little." It increases the productivity of the programmer and the stability of program code, which in turn reduces the stress on the programmer and the time spent on debugging.

Unit testing can be done in two ways − manual testing and automated testing.

|  |  |
| --- | --- |
| **Manual Testing** | **Automated Testing** |
| Executing test cases manually without any tool support is known as manual testing. | Taking tool support and executing the test cases by using an automation tool is known as automation testing. |
| **Time-consuming and tedious** − Since test cases are executed by human resources, it is very slow and tedious. | **Fast** − Automation runs test cases significantly faster than human resources. |
| **Huge investment in human resources** − As test cases need to be executed manually, more testers are required in manual testing. | **Less investment in human resources** − Test cases are executed using automation tools, so less number of testers are required in automation testing. |
| **Less reliable** − Manual testing is less reliable, as it has to account for human errors. | **More reliable** − Automation tests are precise and reliable. |
| **Non-programmable** − No programming can be done to write sophisticated tests to fetch hidden information. | **Programmable** − Testers can program sophisticated tests to bring out hidden information. |

Features of JUnit

* JUnit is an open source framework, which is used for writing and running tests.
* Provides annotations to identify test methods.
* Provides assertions for testing expected results.
* Provides test runners for running tests.
* JUnit tests allow you to write codes faster, which increases quality.
* JUnit is elegantly simple. It is less complex and takes less time.
* JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
* JUnit tests can be organized into test suites containing test cases and even other test suites.
* JUnit shows test progress in a bar that is green if the test is running smoothly, and it turns red when a test fails.

What is a Unit Test Case?

A Unit Test Case is a part of code, which ensures that another part of code (method) works as expected.

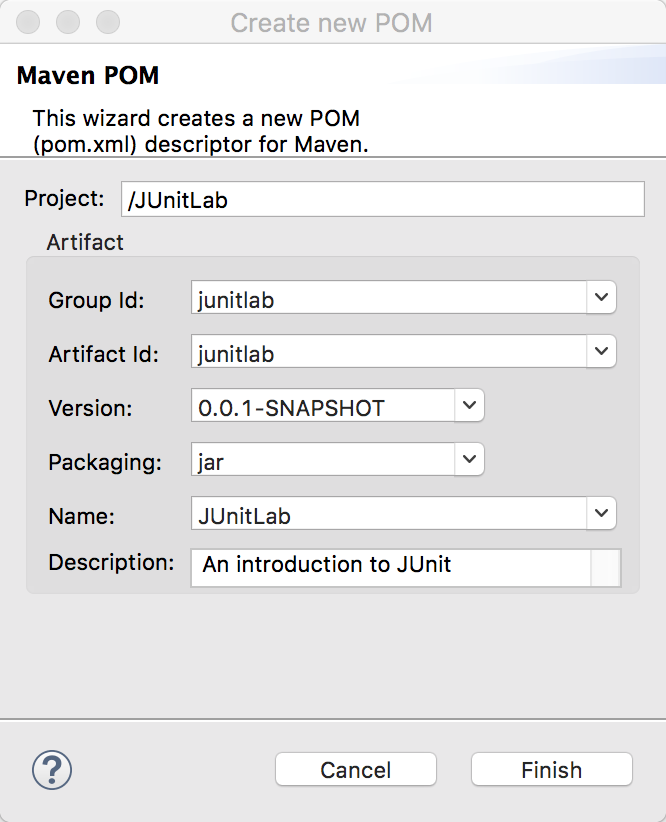
A formal written unit test case is characterized by a known input and an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a post-condition.

There must be at least two unit test cases for each requirement − one positive test and one negative test. If a requirement has sub-requirements, each sub-requirement must have at least two test cases as positive and negative.

## Setting Up an Eclipse Project with Maven Support

Create a new Java project in Eclipse. Select File 🡪 New 🡪 Project. From the dialog window, click the folder for Java and select Java project. Click next and pick a name for your project (JUnitLab). Verify that it’s using Java 1.8 for the execution environment. Click finish. Eclipse may warn you that it needs to open a different perspective. Go ahead and do that.

Now we need to convert it to a Maven project. What is Maven? Maven is a project management tool that is based on POM (project object model). It is used for projects build, dependency and documentation. Maven also maximizes our ability to share code without having to download and install jars on the build path. Instead, Maven links to the online repository that holds all of the jars for potential builds and adds the requested one to the build path. This means that multiple users can use the same pom.xml file for their project and easily be able to maintain the project without constantly adding new resources to the building path. Some organizations will use Ivy or Gradle but the concept is still the same.

Now, we need to add Maven as a build manager for our project. To do so, right click on the project and select **Configure > Convert to Maven project**. The Create New POM dialog appears, enter the following information:

Note that the **Group Id** will be used as the main package for our Java code. It will identify your project uniquely across all projects. It has to follow the package name rules, what means that has to be at least as a domain name you control, and you can create as many subgroups as you want. Since we are not deploying these projects on actual functional webservers, this does not matter much for our practice but I want you to be familiar with the naming. eg. org.apache.maven, com.amazon.cart, edu.dmacc.java2

The **Artifact Id** is the name of the jar without version. If you created it then you can choose whatever name you want with lowercase letters and no strange symbols. If it's a third party jar you have to take the name of the jar as it's distributed. eg. maven, commons-math

If we were working with multiple iterations or builds, the **Version** would update for each change. We won’t see changes in our program, but if you work on a large program, it will be very common for you to update the version along the way (1.0, 1.1, 1.0.1, ...).

Now you see the **pom.xml** file created in the project (in the root of the project folder) and it should open up to the Overview page by default. One of the first things you should do to make your life easier in Java II is to enable Maven index searching. Trust me: it’s easier to search for dependencies than to type them in (incorrectly).

Windows Users: In Eclipse, click on **Windows > Preferences**, and then choose **Maven** from the left side.

Mac users: In Eclipse, click **Eclipse > Preferences** and then choose **Maven** from the left side.

1. Check the box "**Download repository index updates on startup**".
2. check the boxes **Download Artifact Sources** and **Download Artifact JavaDoc**.
3. Click **OK**.
4. Shut down Eclipse and restart it.

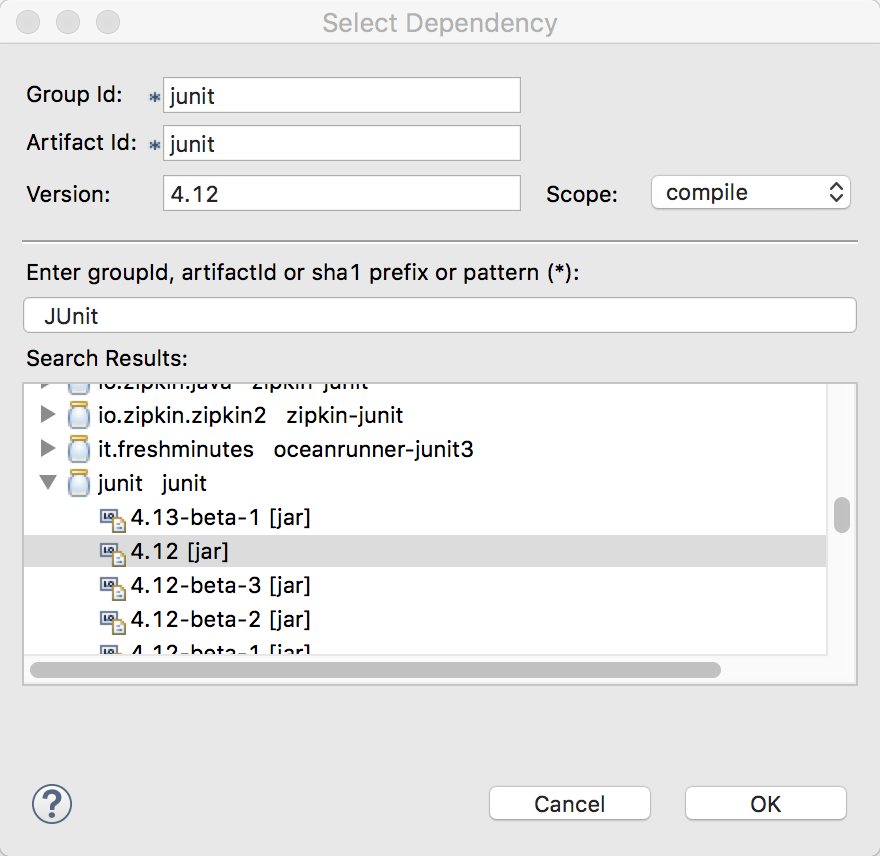
Now, Eclipse will index the Maven repos. THIS TAKES TIME! Be patient. Walk away. Check on it so your computer doesn’t fall asleep. But you’ll see this at the bottom of your Eclipse window.

After they are finished indexing, open back up your project and the pom.xml file.

Using the tabs along the bottom, click on pom.xml to open it in XML mode. This is where you can copy and paste dependencies, but we are going to search for them. Click on the dependencies tab.



Click the add button. In the dialog box, search for JUnit. You’ll receive a lot of results. Scroll through them until you find junit: junit. Click on it to expand the versions.



Select 4.12 from the list and click OK. It’s very important that you’re connected to the Internet so the dependency downloads from the Maven repo site. Save the pom.xml and the dependency will download to your computer. Now you have JUnit support in your project.

## Features of JUnit Test Framework

JUnit test framework provides the following important features −

* Fixtures
* JUnit classes
* Test suites
* Test runners

### Fixtures

**Fixtures** is a fixed state of a set of objects used as a baseline for running tests. The purpose of a test fixture is to ensure that there is a well-known and fixed environment in which tests are run so that results are repeatable. It includes −

* setUp() method, which runs before every test invocation.
* tearDown() method, which runs after every test method.

In Eclipse, create a new JUnit Test Case and call it FixturesExample.

* Go to File 🡪 New 🡪 Other 🡪 Java 🡪 JUnit 🡪 JUnit Test Case
* Make sure New JUnit 4 Test is selected
* Place ‘FixturesExample’ as the name
* Add a setUp( ) method stub
* Press Submit

If you press the green run button now, you’ll see your test fails. This is because of the method in the Test( ) method.

Add the following code to your FixturesExample TestCase:

public class FixturesExample {

protected int value1, value2;

@Before

public void setUp() throws Exception {

value1 = 3;

value2 = 3;

}

@Test

public void test() {

double result = value1 + value2;

*assertTrue*(result == 6);

}

## }

## When you run this test, the setUp( ) method assigns the values for the other tests in the test case to use. We also deleted the fails( ) method so it doesn’t automatically fail.

**Note the @Test and the @Before annotations.** These annotations in JUnit provide the following information about test methods −

* which methods are going to run before and after test methods.
* which methods run before and after all the methods, and.
* which methods or classes will be ignored during the execution.

The following table provides a list of annotations and their meaning in JUnit:

|  |
| --- |
| **Annotation & Description** |
| **@Test**  The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case. |
| **@Before**  Several tests need similar objects created before they can run. Annotating a public void method with @Before causes that method to be run before each Test method. |
| **@After**  If you allocate external resources in a Before method, you need to release them after the test runs. Annotating a public void method with @After causes that method to be run after the Test method. |
| **@BeforeClass**  Annotating a public static void method with @BeforeClass causes it to be run once before any of the test methods in the class. |
| **@AfterClass**  This will perform the method after all tests have finished. This can be used to perform clean-up activities. |
| **@Ignore**  The Ignore annotation is used to ignore the test and that test will not be executed. |

## Before each test we write, we’re going to include the @Test annotation so JUnit knows it’s a test.

## Test Suites

A test suite bundles a few unit test cases and runs them together. In JUnit, both @RunWith and @Suite annotation are used to run the suite test. Let’s create two tests and try it.

Create a new JUnit Test Case and call it TestMessage1. Add in the following code:

public class TestMessage1 {

String message = "Robert";

MessageUtil messageUtil = new MessageUtil(message);

@Before

public void setUp() throws Exception {

}

@Test

public void test() {

System.*out*.println("Inside testPrintMessage()");

assertEquals(message, messageUtil.printMessage());

}

}

Right now, you will have errors because we haven’t created the MessageUtil class. We will. Be patient and ignore those red X’s for now.

I could set my values for my message string in the setUp( ) method. Either way would allow me to use the set value, but the variables have to be set globally so I can use them in all the methods.

Create a new JUnit Test case and call it TestMessage2. Add in the following code:

public class TestMessage2 {

String message = "Robert";

MessageUtil messageUtil = new MessageUtil(message);

@Before

public void setUp() throws Exception {

}

@Test

public void test() {

System.*out*.println("Inside testSalutationMessage()");

message = "Hi!" + "Robert";

assertEquals(message,messageUtil.salutationMessage());

}

}

Before we fix the errors, we’re going to group these two tests together by creating a Test Suite.

Go to Go to File 🡪 New 🡪 Other 🡪 Java 🡪 JUnit 🡪 JUnit Test Suite.

Select TestMessage1 and TestMessage2 and select Finish.

Inside the new class, you’ll see the following code:

@RunWith(Suite.class)

@SuiteClasses({ TestMessage1.class, TestMessage2.class })

public class AllTests {

}

Now, if you run the AllTests class, first off, you’ll get a notice about errors in the project. Just click Proceed. In the JUnit tab, you’ll see that both tests are there (and they are both failing, as they should be). Now, instead of having to run all the tests separately, we can run this one class. It saves a lot of time!

## Red-green Testing

What we have been doing is called red-green testing. We want the tests to come first. When we write a test which needs some functionality, that is a concrete expression of what is needed, and that new bit of functionality is well defined. Initially that functionality does not exist (so the test is red); when we have successfully added the functionality, the test is green. It's a clean determination: either the functionality is present and the test is passing - or it isn't, and the test is failing.

Some developers cannot handle the errors. Fight the urge to immediately fix it.

Implement the Functionality

Let’s create the MessageUtil class. Inside the src/main/java folder, create a new package called model. Inside the model package, create a new class called MessageUtil.

package model;

public class MessageUtil {

private String message;

public MessageUtil(String message) {

this.message = message;

}

// prints the message

public String printMessage() {

System.out.println(message);

return message;

}

## }

## Now, go back into your Tests and import the MessageUtil from the model package.

## Change over to the AllTests class and run your Test Suite. You should find now that the TestMessage1 test passes but the TestMessage2 test does not.

## 

## The second part gives you an explanation of what is wrong.

## 

## So far, we have our tests (TestMessage1, TestMessage2) which we grouped together because they are related (AllTests) but how to we run the tests, aside from pressing the Green button. What if we didn’t have that cool JUnit tab? We would then create a Test Runner to run all our tests.

## Test Runners

A test runner is used for executing the test cases. Inside of the junitsample package, create a TestRunner class with a main method. Add in the following code:

public class TestRunner {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(AllTests.class);

for (Failure failure : result.getFailures()) {

System.out.println(failure.toString());

}

System.out.println(result.wasSuccessful());

}

}

You will have three imports you will need to add. Click the red X to import the appropriate package.

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

## You’ll see we find the same results as the JUnit tab had, but a program could use the results from result.wasSuccessful( ) to determine a next step.

## Finishing Implementation

## Add the saluationMessage( ) method to the MessageUtil class to finish our implementation.

public String salutationMessage() {

return "Hi!" + message;

## }

## Then run your TestRunner to see all the tests pass.

## Assert Class

This class provides a set of assertion methods useful for writing tests in our test cases. Only failed assertions are recorded. Some of the important methods of Assert class are as follows −

|  |
| --- |
| **Methods & Description** |
| **void assertEquals(boolean expected, boolean actual)**  Checks that two primitives/objects are equal. |
| **void assertFalse(boolean condition)**  Checks that a condition is false. |
| **void assertNotNull(Object object)**  Checks that an object isn't null. |
| **void assertNull(Object object)**  Checks that an object is null. |
| **void assertTrue(boolean condition)**  Checks that a condition is true. |
| **void fail()**  Fails a test with no message. |

Let's use some of the above-mentioned methods in an example. Open your TestMessage1 test and add in another test.

@Test

public void testMessageLength() {

System.out.println("Inside testOfMessageLength()");

int messageLength = message.length();

assertTrue(messageLength == messageUtil.returnMessageLength());

}

Run your TestRunner.java and you should have a failure. Let’s implement the returnMessageLength( ) method.

public int returnMessageLength() {

return message.length();

}

Add one more test to make sure the message string isn’t null.

@Test

public void testMessageNotNull() {

System.out.println("Inside testOfMessageNotNull()");

assertNotNull(messageUtil.printMessage( ));

}

## A Larger Example Pulling It All Together - Employee

Here we will see one complete example of JUnit testing using POJO class, Business logic class, and a test class, which will be run by the test runner.

Create a Maven Project (simple project) called employee.

Add in the Maven dependency for JUnit.

Create a package called model in the src/main/java package. Create an **Employee.java** class, which is a POJO class with the following instance variables:

* private String name;
* private double monthlySalary;
* private int age;

Include your getters and setters, a default no-arg constructor and a constructor that takes name and sets the name instance variable.

Create a class called **EmpBusinessLogic.java** in the model package, which contains the business logic but let’s not implement any methods yet. **EmpBusinessLogic** class is used for calculating the yearly salary of an employee and the appraisal amount of an employee.

public class EmpBusinessLogic {

public double calculateYearlySalary(Employee employee) {

return 0.0;

}

public double calculateAppraisal(Employee employee) {

return 0.0;

}

}

Next, create a test case called **TestEmployeeLogic.java** in a tests package. **TestEmployeeLogic** class is used for testing the methods of **EmpBusinessLogic** class. It tests the yearly salary of the employee and tests the appraisal amount of the employee.

public class TestEmployeeLogic {

EmpBusinessLogic empBL = new EmpBusinessLogic();

Employee employee = new Employee("Ryan");

@Before

public void setUp() throws Exception {

}

@Test

public void testCalculateAppraisal() {

employee.setMonthlySalary(8000);

double appraisal = empBL.calculateAppraisal(employee);

*assertEquals*(500, appraisal, 0.0);

}

@Test

public void testCalculateYearlySalary() {

employee.setMonthlySalary(8000);

double salary = empBL.calculateYearlySalary(employee);

*assertEquals*(96000, salary, 0.0);

}

}

Next, create a java class filed named **TestRunner.java** in the test package to execute test case(s).

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner {

public static void main(String[] args) {

Result result = JUnitCore.*runClasses*(TestEmployeeLogic.class);

for (Failure failure : result.getFailures()) {

System.*out*.println(failure.toString());

}

System.*out*.println(result.wasSuccessful());

}

}

Run the Test Runner class and see the failure!

testCalculateAppraisal(TestEmployeeLogic): expected:<500.0> but was:<0.0>

testCalculateYearlySalary(TestEmployeeLogic): expected:<96000.0> but was:<0.0>

false

Now, let’s implement the methods.

public double calculateYearlySalary(Employee employee) {

double yearlySalary = 0;

yearlySalary = employee.getMonthlySalary() \* 12;

return yearlySalary;

}

public double calculateAppraisal(Employee employee) {

double appraisal = 0;

if (employee.getMonthlySalary() < 10000) {

appraisal = 500;

} else {

appraisal = 1000;

}

return appraisal;

}

Run it again and check to see if it passes.

Next, create two more tests:

@Test

public void testEmployeeIsRetirementAge() {

employee.setAge(70);

assertTrue(empBL.isRetirementAge(employee));

}

@Test

public void testEmployeeIsNotRetirementAge() {

employee.setAge(25);

assertFalse(empBL.isRetirementAge(employee));

}

Run your tests runner and watch them fail.

Add an implementation for isRetirementAge( ) in the EmpBusinessLogic class to return true if the employee is older than 65. Otherwise, the employee is not retirement age.

Test your implementation of the method to get all your tests to pass.