**Creating the Project**

To complete this tutorial you first create a Java class library project called JUnit-Sample. After you create the project, you copy two classes from the sample project JUnitSampleSol to your project JUnit-Sample.

**Creating the Java Class Library Project**

1. Choose File > New Project from the main menu.
2. Select Java Class Library from the Java category and click Next.
3. Type **JUnit-Sample** for the project and set the project location.
4. Deselect the Use Dedicated Folder option, if selected.

For this tutorial there is little reason to copy project libraries to a dedicated folder because you will not need to share libraries with other users or projects.

Click Finish.

The first time that you create a JUnit test the IDE prompts you to select a version and then adds a Test Libraries node and the JUnit library.

**Downloading the Solution Project**

You can download the sample project JUnitSampleSol used in this tutorial in the following ways.

* Download [a zip archive of the finished project](https://netbeans.org/projects/samples/downloads/download/Samples/Java/JUnitSampleSol.zip).
* Checkout the project sources from the NetBeans Samples by performing the following steps:
  1. Choose Team > Subversion > Checkout from the main menu.
  2. In the Checkout dialog box, enter the following Repository URL:  
     https://svn.netbeans.org/svn/samples~samples-source-code  
     Click Next.
  3. In the Folders to Checkout panel, click Browse to open the Browse Repository Folders dialog box.
  4. Expand the root node and select **samples/java/JUnitSampleSol**. Click OK.
  5. Specify the Local Folder for the sources. Click Finish.

When you click Finish, the IDE initializes the local folder as a Subversion repository and checks out the project sources.

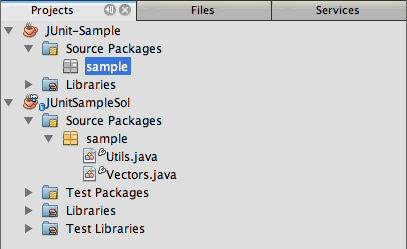
* 1. Click Open Project in the dialog that appears when checkout is complete.

For more about installing Subversion, see the section on [Setting up Subversion](https://netbeans.org/kb/docs/ide/subversion.html#settingUp) in the [Guide to Subversion in NetBeans IDE](https://netbeans.org/kb/docs/ide/subversion.html).

**Note.** If you did not install the JUnit plugin when you installed the IDE, when you open the NetBeans project you will be prompted to install the JUnit plugin to resolve the reference to the JUnit libraries.

**Creating the Java Classes**

In this exercise you copy the files Utils.java and Vectors.java from the sample project JUnitSampleSol into the class library project that you created.

1. In the Projects window, right-click the Source Packages node of the **JUnit-Sample** project and choose New > Java Package from the popup menu.
2. Type **sample** as the package name. Click Finish.
3. Open the **JUnitSampleSol** project (if not already open) and expand the Source Packages node in the Projects window.  
   
4. Copy the classes Utils.java and Vectors.java in the JUnitSampleSol project and paste them into the sample source package in JUnit-Sample.

If you look at the source code for the classes, you can see that Utils.java has three methods (computeFactorial, concatWords, and normalizeWord) and that Vectors.java has two methods (equal and scalarMultiplication). The next step is to create test classes for each class and write some test cases for the methods.

**Note.** You can close the JUnitSampleSol project because you will not need it again. The JUnitSampleSol project contains all the tests described in this document.

**Writing JUnit 3 Unit Tests**

In this part of the tutorial you create basic JUnit 3 unit tests for the classes Vectors.java and Utils.java. You will use the IDE to create skeleton test classes that are based on the classes in your project. You will then modify the generated test methods and add new test methods.

The IDE prompts you to choose a JUnit version the first time that you use the IDE to create tests for you in the project. The version that you select becomes the default JUnit version and the IDE will generate all subsequent tests and test suites for that version.

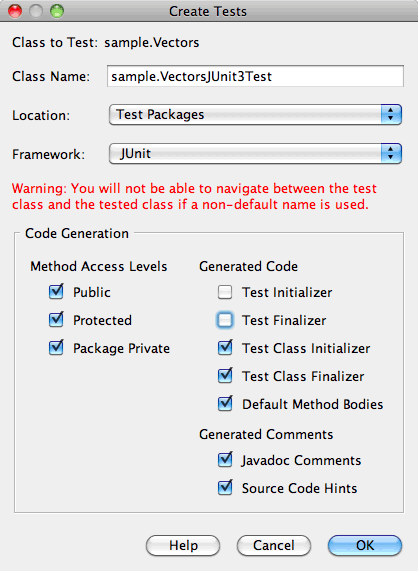
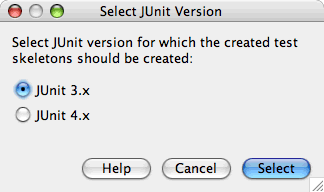
**Creating a Test Class for Vectors.java**

In this exercise you create a JUnit test skeleton for Vectors.java. You will also select JUnit as the test framework and JUnit 3 as the version.

**Note.** If you are using NetBeans IDE 7.1 or earlier you do not need to specify the test framework because JUnit is specified by default. In NetBeans IDE 7.2 you have the option of specifying JUnit or TestNG as the test framework.

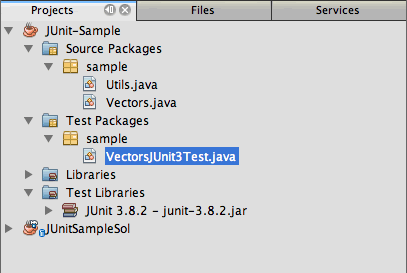
1. Right-click Vectors.java and choose Tools > Create Tests.
2. Modify the name of the test class to **VectorsJUnit3Test** in the Create Tests dialog.

When you change the name of the test class, you will see a warning about changing the name. The default name is based on the name of the class you are testing, with the word Test appended to the name. For example, for the class MyClass.java, the default name of the test class is MyClassTest.java. Usually it is best to keep the default name, but for this tutorial you will change the name because you will also create JUnit 4 tests in the same package and the names of the test classes must be unique.

1. Select JUnit in the Framework dropdown list.
2. Deselect Test Initializer and Test Finalizer. Click OK.  
   
3. Select JUnit 3.x in the Select JUnit Version dialog box.  
   

When you select JUnit 3.x the IDE adds the JUnit 3 library to the project.

When you click Select, the IDE creates the VectorsJUnit3Test.java test class in the sample package under the Test Packages node in the Projects window.



A project requires a directory for test packages to create tests. The default location for the test packages directory is at the root level of the project, but depending on the type of project you can specify a different location for the directory in the project's Properties dialog.

If you look at the generated test class VectorsJUnit3Test.java in the editor, you can see that the IDE generated the following test class with test methods for the methods equal and scalarMultiplication.

public class VectorsJUnit3Test extends TestCase {

/\*\*

\* Test of equal method, of class Vectors.

\*/

public void testEqual() {

System.out.println("equal");

int[] a = null;

int[] b = null;

boolean expResult = false;

boolean result = Vectors.equal(a, b);

assertEquals(expResult, result);

// TODO review the generated test code and remove the default call to fail.

fail("The test case is a prototype.");

}

/\*\*

\* Test of scalarMultiplication method, of class Vectors.

\*/

public void testScalarMultiplication() {

System.out.println("scalarMultiplication");

int[] a = null;

int[] b = null;

int expResult = 0;

int result = Vectors.scalarMultiplication(a, b);

assertEquals(expResult, result);

// TODO review the generated test code and remove the default call to fail.

fail("The test case is a prototype.");

}

}

The method body of each generated test is provided solely as a guide and needs to be modified to be an actual test case. You can deselect Default Method Bodies in the Create Tests dialog if you do not want the code generated for you.

When the IDE generates the names for the test methods, each method name is prepended with test because JUnit 3 uses naming conventions and reflection to identify tests. To identify test methods, each test method is required to follow the syntax test*<NAME>*.

**Note.** In JUnit 4 it is no longer necessary to use this test method naming syntax because you can use annotations to identify test methods and the test class is no longer required to extend TestCase.

**Writing Test Methods for Vectors.java**

In this exercise you modify the generated test methods to make them functioning tests and modify the default output messages. You do not need to modify the output messages to run the tests, but you may want to modify the output to help identify the results displayed in the JUnit Test Results output window.

1. Open VectorsJUnit3Test.java in the editor.
2. Modify the test skeleton for testScalarMultiplication by changing the value of the println and removing the generated variables. The test method should now look like the following (changes displayed in bold):
3. public void testScalarMultiplication() {
4. System.out.println("**\* VectorsJUnit3Test: testScalarMultiplication()**");
5. assertEquals(expResult, result);

}

1. Now add some assertions to test the method.
2. public void testScalarMultiplication() {
3. System.out.println("\* VectorsJUnit3Test: testScalarMultiplication()");
4. **assertEquals( 0, Vectors.scalarMultiplication(new int[] { 0, 0}, new int[] { 0, 0}));**
5. **assertEquals( 39, Vectors.scalarMultiplication(new int[] { 3, 4}, new int[] { 5, 6}));**
6. **assertEquals(-39, Vectors.scalarMultiplication(new int[] {-3, 4}, new int[] { 5,-6}));**
7. **assertEquals( 0, Vectors.scalarMultiplication(new int[] { 5, 9}, new int[] {-9, 5}));**
8. **assertEquals(100, Vectors.scalarMultiplication(new int[] { 6, 8}, new int[] { 6, 8}));**

}

This test method uses the JUnit assertEquals method. To use the assertion, you supply the input variables and the expected result. To pass the test, the test method must successfully return all the expected results based on the supplied variables when running the tested method. You should add a sufficient number of assertions to cover the various possible permutations.

1. Modify the test skeleton for testEqual by deleting the generated method bodies and adding the following println.

**System.out.println("\* VectorsJUnit3Test: testEqual()");**

The test method should now look like the following:

public void testEqual() {

System.out.println("\* VectorsJUnit3Test: testEqual()");

}

1. Modify the testEqual method by adding the following assertions (displayed in bold).
2. public void testEqual() {
3. System.out.println("\* VectorsJUnit3Test: testEqual()");
4. **assertTrue(Vectors.equal(new int[] {}, new int[] {}));**
5. **assertTrue(Vectors.equal(new int[] {0}, new int[] {0}));**
6. **assertTrue(Vectors.equal(new int[] {0, 0}, new int[] {0, 0}));**
7. **assertTrue(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0, 0}));**
8. **assertTrue(Vectors.equal(new int[] {5, 6, 7}, new int[] {5, 6, 7}));**
9. **assertFalse(Vectors.equal(new int[] {}, new int[] {0}));**
10. **assertFalse(Vectors.equal(new int[] {0}, new int[] {0, 0}));**
11. **assertFalse(Vectors.equal(new int[] {0, 0}, new int[] {0, 0, 0}));**
12. **assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0}));**
13. **assertFalse(Vectors.equal(new int[] {0, 0}, new int[] {0}));**
14. **assertFalse(Vectors.equal(new int[] {0}, new int[] {}));**
15. **assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0, 1}));**
16. **assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 1, 0}));**
17. **assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {1, 0, 0}));**
18. **assertFalse(Vectors.equal(new int[] {0, 0, 1}, new int[] {0, 0, 3}));**

}

This test uses the JUnit assertTrue and assertFalse methods to test a variety of possible results. For the test of this method to pass, the assertTrue must all be true and assertFalse must all be false.

1. Save your changes.

Compare: [Writing Test Methods for Vectors.java (JUnit 4)](https://netbeans.org/kb/docs/java/junit-intro.html#Exercise_32)

**Creating a Test Class for Utils.java**

You now create the test skeletons for Utils.java. When you created the test in the previous exercise, the IDE prompted you for the version of JUnit. You are not prompted to select a version this time.

1. Right-click Utils.java and choose Tools > Create Tests.
2. Select JUnit in the Framework dropdown list if not selected.
3. Select Test Initializer and Test Finalizer in the dialog box, if not selected.
4. Modify the name of the test class to **UtilsJUnit3Test** in the Create Tests dialog box. Click OK.

When you click OK, the IDE creates the test file UtilsJUnit3Test.java in the Test Packages > samples directory. You can see that in addition to creating the test skeletons testComputeFactorial, testConcatWords, and testNormalizeWord for the methods in Utils.java, the IDE also creates the test initializer method setUp and the test finalizer method tearDown.

**Writing Test Methods for Utils.java**

In this exercise you add some test cases that illustrate some common JUnit test elements. You also add a println to the methods because some methods do not print any output by default. By adding a println to the methods you can later look in the JUnit test result window to see if the methods were run and the order in which they were run.

**Test Initializers and Finalizers**

The setUp and tearDown methods are used to initialize and finalize test conditions. You do not need the setUp and tearDown methods to test Utils.java, but they are included here to demonstrate how they work.

The setUp method is a test initialization method and is run before each test case in the test class. A test initialization method is not required for running tests, but if you need to initialize some variables before you run a test, you use the test initializer method.

The tearDown method is a test finalizer method and is run after each test case in the test class. A test finalizer method is not required for running tests, but you may need a finalizer to clean up any data that was required when running the test cases.

1. Make the following changes (displayed in bold) to add a println to each method.
2. @Override
3. protected void setUp() throws Exception {
4. super.setUp();
5. **System.out.println("\* UtilsJUnit3Test: setUp() method");**
6. }
7. @Override
8. protected void tearDown() throws Exception {
9. super.tearDown();
10. **System.out.println("\* UtilsJUnit3Test: tearDown() method");**

}

When you run the test the println text for each methods will appear in the JUnit Test Results output window. If you do not add the println, there is no output to show that the methods were run.

**Testing Using a Simple Assertion**

This simple test case tests the concatWords method. Instead of using the generated test method testConcatWords, you will add a new test method called testHelloWorld that uses a single simple assertion to test if the method concatenates the strings correctly. The assertEquals in the test case uses the syntax assertEquals(*EXPECTED\_RESULT, ACTUAL\_RESULT*) to test if the expected result is equal to the actual result. In this case, if the input to the method concatWords is "Hello", ", ", "world" and "!", the expected result should equal "Hello, world!".

1. Delete the generated test method testConcatWords in UtilsJUnit3Test.java.
2. Add the following method to test the concatWords method.
3. **public void testHelloWorld() {**
4. **assertEquals("Hello, world!", Utils.concatWords("Hello", ", ", "world", "!"));**

**}**

1. Add a println statement to display text about the test in the JUnit Test Results window.
2. public void testHelloWorld() {
3. **System.out.println("\* UtilsJUnit3Test: test method 1 - testHelloWorld()");**

assertEquals("Hello, world!", Utils.concatWords("Hello", ", ", "world", "!"));

Compare: [Testing Using a Simple Assertion (JUnit 4)](https://netbeans.org/kb/docs/java/junit-intro.html#Exercise_342)

**Testing Using a Timeout**

This test demonstrates how to check if a method is taking too long to complete. If the method is taking too long, the test thread is interrupted and the test fails. You can specify the time limit in the test.

The test method invokes the computeFactorial method in Utils.java. You can assume that the computeFactorial method is correct, but in this case you want to test if the computation is completed within 1000 milliseconds. The computeFactorial thread and a test thread are started at the same time. The test thread will stop after 1000 milliseconds and throw a TimeoutException unless the computeFactorial thread completes first. You will add a message so that a message is displayed if a TimeoutException is thrown.

1. Delete the generated test method testComputeFactorial.
2. Add the testWithTimeout method that calculates the factorial of a randomly generated number.
3. **public void testWithTimeout() throws InterruptedException, TimeoutException {**
4. **final int factorialOf = 1 + (int) (30000 \* Math.random());**
5. **System.out.println("computing " + factorialOf + '!');**
6. **Thread testThread = new Thread() {**
7. **public void run() {**
8. **System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf));**
9. **}**
10. **};**

**}**

1. Fix your imports to import java.util.concurrent.TimeoutException.
2. Add the following code (displayed in bold) to the method to interrupt the thread and display a message if the test takes too long to execute.
3. Thread testThread = new Thread() {
4. public void run() {
5. System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf));
6. }
7. };
8. **testThread.start();**
9. **Thread.sleep(1000);**
10. **testThread.interrupt();**
11. **if (testThread.isInterrupted()) {**
12. **throw new TimeoutException("the test took too long to complete");**
13. **}**

}

You can modify the Thread.sleep line to change the number of milliseconds before the timeout is thrown.

1. Add the following println (displayed in bold) to print the text about the test in the JUnit Test Results window.
2. public void testWithTimeout() throws InterruptedException, TimeoutException {
3. **System.out.println("\* UtilsJUnit3Test: test method 2 - testWithTimeout()");**
4. final int factorialOf = 1 + (int) (30000 \* Math.random());
5. System.out.println("computing " + factorialOf + '!');

Compare: [Testing Using a Timeout (JUnit 4)](https://netbeans.org/kb/docs/java/junit-intro.html#Exercise_343)

**Testing for an Expected Exception**

This test demonstrates how to test for an expected exception. The method fails if it does not throw the specified expected exception. In this case you are testing that the computeFactorial method throws an IllegalArgumentException if the input variable is a negative number (-5).

1. Add the following testExpectedException method that invokes the computeFactorial method with an input of -5.
2. **public void testExpectedException() {**
3. **try {**
4. **final int factorialOf = -5;**
5. **System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf));**
6. **fail("IllegalArgumentException was expected");**
7. **} catch (IllegalArgumentException ex) {**
8. **}**

**}**

1. Add the following println (displayed in bold) to print the text about the test in the JUnit Test Results window.
2. public void testExpectedException() {
3. **System.out.println("\* UtilsJUnit3Test: test method 3 - testExpectedException()");**

try {

Compare: [Testing for an Expected Exception (JUnit 4)](https://netbeans.org/kb/docs/java/junit-intro.html#Exercise_344)

**Disabling a Test**

This test demonstrates how to temporarily disable a test method. In JUnit 3, if a method name does not start with test it is not recognized as a test method. In this case you prepend DISABLED\_ to the name of the test method to disable it.

1. Delete the generated test method testNormalizeWord.
2. Add the following test method to the test class.
3. **public void testTemporarilyDisabled() throws Exception {**
4. **System.out.println("\* UtilsJUnit3Test: test method 4 - checkExpectedException()");**
5. **assertEquals("Malm\u00f6", Utils.normalizeWord("Malmo\u0308"));**

**}**

The test method testTemporarilyDisabled will run if you run the test class.

1. Prepend DISABLED\_ (displayed in bold) to the name of the test method.
2. public void **DISABLED\_**testTemporarilyDisabled() throws Exception {
3. System.out.println("\* UtilsJUnit3Test: test method 4 - checkExpectedException()");
4. assertEquals("Malm\u00f6", Utils.normalizeWord("Malmo\u0308"));

}

Compare: [Disabling a Test (JUnit 4)](https://netbeans.org/kb/docs/java/junit-intro.html#Exercise_345)

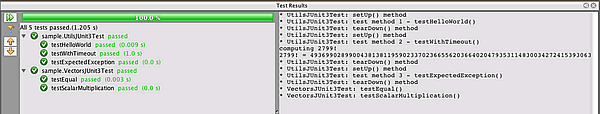
Now that you have written the tests, you can run the test and see the test output in the JUnit Test Results window.

**Running the Tests**

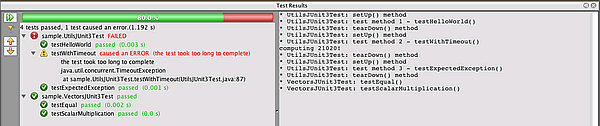
When you run a JUnit test the results are displayed in the Test Results window of the IDE. You can run individual JUnit test classes or you can choose Run > Test *PROJECT\_NAME* from the main menu to run all the tests for the project. If you choose Run > Test, the IDE runs all the test classes in the Test Packages folder. To run an individual test class, right-click the test class under the Test Packages node and choose Run File.

1. Choose Run > Set Main Project in the main menu and select the JUnit-Sample project.
2. Choose Run > Test Project (JUnit-Sample) from the main menu.
3. Choose Window > IDE Tools > Test Results to open the Test Results window.

When you run the test you will see one of the following results in the JUnit Test Results window.

[](https://netbeans.org/images_www/articles/72/java/junit/junit3-test-pass.png)

In this image (click the image to see a larger image) you can see that the project passed all the tests. The left pane displays the results of the individual test methods and the right pane displays the test output. If you look at the output you can see the order that the tests were run. The println that you added to each of the test methods printed out the name of the test to the output window. You can also see that in UtilJUnit3Test the setUp method was run before each test method and the tearDown method was run after each method.

[](https://netbeans.org/images_www/articles/72/java/junit/junit3-test-fail.png)

In this image (click the image to see a larger image) you can see that the project failed one of the tests. The testTimeout method took too long to complete and the test thread was interrupted, causing that test to fail. It took longer than 1000 milliseconds to compute the factorial of the randomly generated number (22991).

The next step after you create your unit test classes is to create test suites. See [Creating JUnit 3 Test Suites](https://netbeans.org/kb/docs/java/junit-intro.html#Exercise_41) to see how to run specified tests as a group so you do not have to run each test individually.

**Writing JUnit 4 Tests**

In this exercise you create JUnit 4 unit tests for the classes Vectors.java and Utils.java. The JUnit 4 test cases are the same as the JUnit 3 test cases, but you will see that the syntax for writing the tests is simpler.

You will use the IDE's wizards to create test skeletons based on the classes in your project. The first time that you use the IDE to create some test skeletons for you, the IDE prompts you to choose the JUnit version.

**Note.** If you already selected JUnit 3.x as the default version for your tests, you need to change the default version to JUnit 4.x. To change the default JUnit version, expand the Test Libraries node, right-click the JUnit library and choose Remove. You can now use the Add Library dialog box to explicitly add the JUnit 4 library or you can select version 4.x when you are prompted to select the JUnit version when you create a new test. You can still run JUnit 3 tests, but any new tests you create will use JUnit 4.

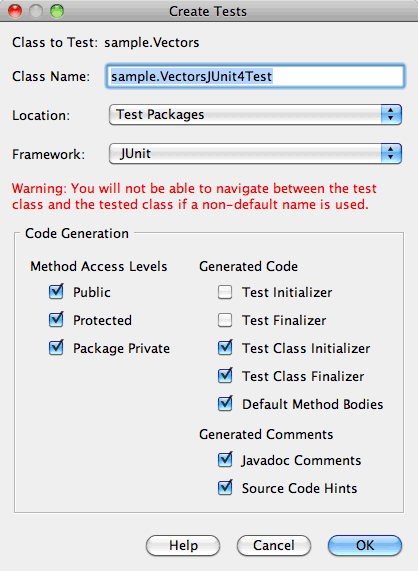
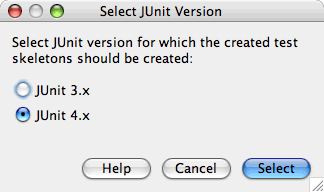
**Creating a Test Class for Vectors.java**

In this exercise you will create the JUnit test skeletons for Vectors.java.

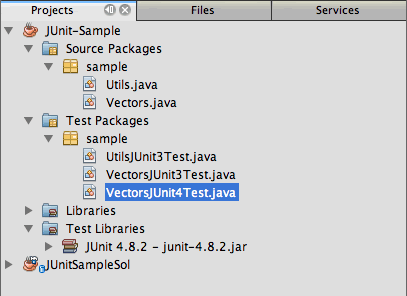
**Note.** If you are using NetBeans IDE 7.1 or earlier you do not need to specify the test framework because JUnit is specified by default. In NetBeans IDE 7.2 you have the option of specifying JUnit or TestNG as the test framework.

1. Right-click Vectors.java and choose Tools > Create Tests.
2. Modify the name of the test class to **VectorsJUnit4Test** in the Create Tests dialog.

When you change the name of the test class, you will see a warning about changing the name. The default name is based on the name of the class you are testing, with the word Test appended to the name. For example, for the class MyClass.java, the default name of the test class is MyClassTest.java. Unlike JUnit 3, in JUnit 4, test are not required to end with the word Test. Usually it is best to keep the default name, but because you are creating all the JUnit tests in the same package in this tutorial the names of the test classes have to be unique.

1. Select JUnit in the Framework dropdown list.
2. Deselect Test Initializer and Test Finalizer. Click OK.  
   
3. Select JUnit 4.x in the Select JUnit Version dialog box. Click Select.  
   

When you click OK, the IDE creates the VectorsJUnit4Test.java test class in the sample package under the Test Packages node in the Projects window.



**Note.** A project requires a directory for test packages to create tests. The default location for the test packages directory is at the root level of the project, but you can specify a different location for the directory in the project's Properties dialog.

If you look at VectorsJUnit3Test.java in the editor, you can see that the IDE generated the test methods testEqual and testScalarMultiplication. In VectorsJUnit4Test.java, each test method is annotated with @Test. The IDE generated the names for the test methods based on the names of the method in Vectors.java but the name of the test method is not required to have test prepended. The default body of each generated test method is provided solely as a guide and needs to be modified to be actual test cases.

You can deselect Default Method Bodies in the Create Tests dialog if you do not want the bodies of the method generated for you.

The IDE also generated the following test class initializer and finalizer methods:

@BeforeClass

public static void setUpClass() throws Exception {

}

@AfterClass

public static void tearDownClass() throws Exception {

}

The IDE generates the class initializer and finalizer methods by default when creating JUnit 4 test classes. The annotations @BeforeClass and @AfterClass are used to mark methods that should be run before and after running the test class. You can delete the methods because you will not need them to test Vectors.java.

You can configure the methods that are generated by default by configuring the JUnit options in the Options window.

**Note.** For JUnit 4 tests, notice that by default the IDE adds a static import declaration for org.junit.Assert.\*.

**Writing Test Methods for Vectors.java**

In this exercise you modify each of the generated test methods to test the methods using the JUnit assert method and to change the names of the test methods. In JUnit 4 you have greater flexibility when naming test methods because test methods are indicated by the @Test annotation and do not require the word test prepended to test method names.

1. Open VectorsJUnit4Test.java in the editor.
2. Modify the test method for testScalarMultiplication by changing the name of the method, the value of the println and removing the generated variables. The test method should now look like the following (changes displayed in bold):
3. @Test
4. public void **ScalarMultiplicationCheck**() {
5. System.out.println("**\* VectorsJUnit4Test: ScalarMultiplicationCheck()**");
6. assertEquals(expResult, result);

}

**Note.** When writing tests it is not necessary to change the printed output. You do this in this exercise so that it is easier to identify the test results in the output window.

1. Now add some assertions to test the method.
2. @Test
3. public void ScalarMultiplicationCheck() {
4. System.out.println("\* VectorsJUnit4Test: ScalarMultiplicationCheck()");
5. **assertEquals( 0, Vectors.scalarMultiplication(new int[] { 0, 0}, new int[] { 0, 0}));**
6. **assertEquals( 39, Vectors.scalarMultiplication(new int[] { 3, 4}, new int[] { 5, 6}));**
7. **assertEquals(-39, Vectors.scalarMultiplication(new int[] {-3, 4}, new int[] { 5,-6}));**
8. **assertEquals( 0, Vectors.scalarMultiplication(new int[] { 5, 9}, new int[] {-9, 5}));**
9. **assertEquals(100, Vectors.scalarMultiplication(new int[] { 6, 8}, new int[] { 6, 8}));**

}

In this test method you use the JUnit assertEquals method. To use the assertion, you supply the input variables and the expected result. To pass the test, the test method must successfully return all the expected results based on the supplied variables when running the tested method. You should add a sufficient number of assertions to cover the various possible permutations.

1. Change the name of the testEqual test method to equalsCheck.
2. Delete the the generated method body of the equalsCheck test method.
3. Add the following println to the equalsCheck test method.

**System.out.println("\* VectorsJUnit4Test: equalsCheck()");**

The test method should now look like the following:

@Test

public void equalsCheck() {

System.out.println("\* VectorsJUnit4Test: equalsCheck()");

}

1. Modify the equalsCheck method by adding the following assertions (displayed in bold).
2. @Test
3. public void equalsCheck() {
4. System.out.println("\* VectorsJUnit4Test: equalsCheck()");
5. **assertTrue(Vectors.equal(new int[] {}, new int[] {}));**
6. **assertTrue(Vectors.equal(new int[] {0}, new int[] {0}));**
7. **assertTrue(Vectors.equal(new int[] {0, 0}, new int[] {0, 0}));**
8. **assertTrue(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0, 0}));**
9. **assertTrue(Vectors.equal(new int[] {5, 6, 7}, new int[] {5, 6, 7}));**
10. **assertFalse(Vectors.equal(new int[] {}, new int[] {0}));**
11. **assertFalse(Vectors.equal(new int[] {0}, new int[] {0, 0}));**
12. **assertFalse(Vectors.equal(new int[] {0, 0}, new int[] {0, 0, 0}));**
13. **assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0}));**
14. **assertFalse(Vectors.equal(new int[] {0, 0}, new int[] {0}));**
15. **assertFalse(Vectors.equal(new int[] {0}, new int[] {}));**
16. **assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0, 1}));**
17. **assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 1, 0}));**
18. **assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {1, 0, 0}));**
19. **assertFalse(Vectors.equal(new int[] {0, 0, 1}, new int[] {0, 0, 3}));**

}

This test uses the JUnit assertTrue and assertFalse methods to test a variety of possible results. For the test of this method to pass, the assertTrue must all be true and assertFalse must all be false.

Compare: [Writing Test Methods for Vectors.java (JUnit 3)](https://netbeans.org/kb/docs/java/junit-intro.html#Exercise_22)

**Creating a Test Class for Utils.java**

You will now create the JUnit test methods for Utils.java. When you created the test class in the previous exercise, the IDE prompted you for the version of JUnit. You are not prompted to select a version this time because you already selected the JUnit version and all subsequent JUnit tests are created in that version.

**Note.** You can still write and run JUnit 3 tests if you select JUnit 4 as the version, but the IDE uses the JUnit 4 template for generating test skeletons.

1. Right-click Utils.java and choose Tools > Create Tests.
2. Select JUnit in the Framework dropdown list if not selected.
3. Select Test Initializer and Test Finalizer in the dialog box if not selected.
4. Modify the name of the test class to **UtilsJUnit4Test** in the Create Tests dialog box. Click OK.

When you click OK, the IDE creates the test file UtilsJUnit4Test.java in the Test Packages > sample directory. You can see that the IDE generated the test methods testComputeFactorial, testConcatWords, and testNormalizeWord for the methods in Utils.java. The IDE also generated initializer and finalizer methods for the test and the test class.

**Writing Test Methods for Utils.java**

In this exercise you will add test cases that illustrate some common JUnit test elements. You will also add a println to the methods because some methods do not print any output to the JUnit Test Results window to indicate that they were run, or to indicate that the method passed the test. By adding a println to the methods you can see if the methods were run and the order in which they were run.

**Test Initializers and Finalizers**

When you created the test class for Utils.java the IDE generated annotated initializer and finalizer methods. You can choose any name for the name of the method because there is no required naming convention.

**Note.** You do not need the initializer and finalizer methods to test Utils.java, but they are included in this tutorial to demonstrate how they work.

In JUnit 4 you can use annotations to mark the following types of initializer and finalizer methods.

* **Test Class Initializer.** The @BeforeClass annotation marks a method as a test class initialization method. A test class initialization method is run only once, and before any of the other methods in the test class. For example, instead of creating a database connection in a test initializer and creating a new connection before each test method, you may want to use a test class initializer to open a connection before running the tests. You could then close the connection with the test class finalizer.
* **Test Class Finalizer.** The @AfterClass annotation marks a method as a test class finalizer method. A test class finalizer method is run only once, and after all of the other methods in the test class are finished.
* **Test Initializer.** The @Before annotation marks a method as a test initialization method. A test initialization method is run before each test case in the test class. A test initialization method is not required to run tests, but if you need to initialize some variables before you run a test, you use a test initializer method.
* **Test Finalizer.** The @After annotation marks a method as a test finalizer method. A test finalizer method is run after each test case in the test class. A test finalizer method is not required to run tests, but you may need a finalizer to clean up any data that was required when running the test cases.

Make the following changes (displayed in bold) in UtilsJUnit4Test.java.

@BeforeClass

public static void setUpClass() throws Exception {

**System.out.println("\* UtilsJUnit4Test: @BeforeClass method");**

}

@AfterClass

public static void tearDownClass() throws Exception {

**System.out.println("\* UtilsJUnit4Test: @AfterClass method");**

}

@Before

public void setUp() {

**System.out.println("\* UtilsJUnit4Test: @Before method");**

}

@After

public void tearDown() {

**System.out.println("\* UtilsJUnit4Test: @After method");**

}

Compare: [Test initializers and finalizers (JUnit 3)](https://netbeans.org/kb/docs/java/junit-intro.html#Exercise_241)

When you run the test class the println text you added is displayed in the output pane of the JUnit Test Results window. If you do not add the println, there is no output to indicate that the initializer and finalizer methods were run.

**Testing Using a Simple Assertion**

This simple test case tests the concatWords method. Instead of using the generated test method testConcatWords, you will add a new test method called helloWorldCheck that uses a single simple assertion to test if the method concatenates the strings correctly. The assertEquals in the test case uses the syntax assertEquals(*EXPECTED\_RESULT, ACTUAL\_RESULT*) to test if the expected result is equal to the actual result. In this case, if the input to the method concatWords is "Hello", ",", "world" and "!", the expected result should equal "Hello, world!".

1. Delete the generated test method testConcatWords.
2. Add the following helloWorldCheck method to test Utils.concatWords.
3. **@Test**
4. **public void helloWorldCheck() {**
5. **assertEquals("Hello, world!", Utils.concatWords("Hello", ", ", "world", "!"));**

**}**

1. Add a println statement to display text about the test in the JUnit Test Results window.
2. @Test
3. public void helloWorldCheck() {
4. **System.out.println("\* UtilsJUnit4Test: test method 1 - helloWorldCheck()");**

assertEquals("Hello, world!", Utils.concatWords("Hello", ", ", "world", "!"));

Compare: [Testing Using a Simple Assertion (JUnit 3)](https://netbeans.org/kb/docs/java/junit-intro.html#Exercise_242)

**Testing Using a Timeout**

This test demonstrates how to check if a method is taking too long to complete. If the method is taking too long, the test thread is interrupted and the test fails. You can specify the time limit in the test.

The test method invokes the computeFactorial method in Utils.java. You can assume that the computeFactorial method is correct, but in this case you want to test if the computation is completed within 1000 milliseconds. You do this by interrupting the test thread after 1000 milliseconds. If the thread is interrupted the test method throws a TimeoutException.

1. Delete the generated test method testComputeFactorial.
2. Add the testWithTimeout method that calculates the factorial of a randomly generated number.
3. **@Test**
4. **public void testWithTimeout() {**
5. **final int factorialOf = 1 + (int) (30000 \* Math.random());**
6. **System.out.println("computing " + factorialOf + '!');**
7. **System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf));**

**}**

1. Add the following code (displayed in bold) to set the timeout and to interrupt the thread if the method takes too long to execute.
2. @Test**(timeout=1000)**
3. public void testWithTimeout() {

final int factorialOf = 1 + (int) (30000 \* Math.random());

You can see that the timeout is set to 1000 milliseconds.

1. Add the following println (displayed in bold) to print the text about the test in the JUnit Test Results window.
2. @Test(timeout=1000)
3. public void testWithTimeout() {
4. **System.out.println("\* UtilsJUnit4Test: test method 2 - testWithTimeout()");**
5. final int factorialOf = 1 + (int) (30000 \* Math.random());
6. System.out.println("computing " + factorialOf + '!');

Compare: [Testing Using a Timeout (JUnit 3)](https://netbeans.org/kb/docs/java/junit-intro.html#Exercise_243)

**Testing for an Expected Exception**

This test demonstrates how to test for an expected exception. The method fails if it does not throw the specified expected exception. In this case you are testing that the computeFactorial method throws an IllegalArgumentException if the input variable is a negative number (-5).

1. Add the following testExpectedException method that invokes the computeFactorial method with an input of -5.
2. **@Test**
3. **public void checkExpectedException() {**
4. **final int factorialOf = -5;**
5. **System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf));**

**}**

1. Add the following property (displayed in bold) to the @Test annotation to specify that the test is expected to throw IllegalArgumentException.
2. @Test**(expected=IllegalArgumentException.class)**
3. public void checkExpectedException() {
4. final int factorialOf = -5;
5. System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf));

}

1. Add the following println (displayed in bold) to print the text about the test in the JUnit Test Results window.
2. @Test (expected=IllegalArgumentException.class)
3. public void checkExpectedException() {
4. **System.out.println("\* UtilsJUnit4Test: test method 3 - checkExpectedException()");**
5. final int factorialOf = -5;
6. System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf));

}

Compare: [Testing for an Expected Exception (JUnit 3)](https://netbeans.org/kb/docs/java/junit-intro.html#Exercise_244)

**Disabling a Test**

This test demonstrates how to temporarily disable a test method. In JUnit 4 you simply add the @Ignore annotation to disable the test.

1. Delete the generated test method testNormalizeWord.
2. Add the following test method to the test class.
3. **@Test**
4. **public void temporarilyDisabledTest() throws Exception {**
5. **System.out.println("\* UtilsJUnit4Test: test method 4 - checkExpectedException()");**
6. **assertEquals("Malm\u00f6", Utils.normalizeWord("Malmo\u0308"));**

**}**

The test method temporarilyDisabledTest will run if you run the test class.

1. Add the @Ignore annotation (displayed in bold) above @Test to disable the test.
2. **@Ignore**
3. @Test
4. public void temporarilyDisabledTest() throws Exception {
5. System.out.println("\* UtilsJUnit4Test: test method 4 - checkExpectedException()");
6. assertEquals("Malm\u00f6", Utils.normalizeWord("Malmo\u0308"));

}

1. Fix your imports to import org.junit.Ignore.

Compare: [Disabling a Test (JUnit 3)](https://netbeans.org/kb/docs/java/junit-intro.html#Exercise_245)

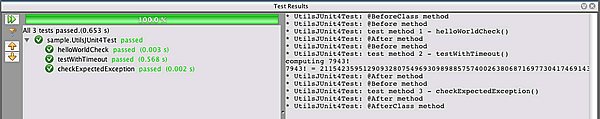
Now that you have written the tests you can run the test and see the test output in the JUnit Test Results window.

**Running the Tests**

You can run JUnit tests on the entire application or on individual files and see the results in the IDE. The easiest way to run all the unit tests for the project is to choose Run > Test *<PROJECT\_NAME>* from the main menu. If you choose this method, the IDE runs all the test classes in the Test Packages. To run an individual test class, right-click the test class under the Test Packages node and choose Run File.

1. Right-click UtilsJUnit4Test.java in the Projects window.
2. Choose Test File.
3. Choose Window > IDE Tools > Test Results to open the Test Results window.

When you run UtilsJUnit4Test.java the IDE only runs the tests in the test class. If the class passes all the tests you will see something similar to the following image in the JUnit Test Results window.

[](https://netbeans.org/images_www/articles/72/java/junit/junit4-utilstest-pass.png)

In this image (click the image to see a larger image) you can see that the IDE ran the JUnit test on Utils.java and that the class passed all the tests. The left pane displays the results of the individual test methods and the right pane displays the test output. If you look at the output you can see the order that the tests were run. The println that you added to each of the test methods printed out the name of the test to Test Results window and the Output window.

You can see that in UtilsJUnit4Test the test class initializer method annotated with @BeforeClass was run before any of the other methods and it was run only once. The test class finalizer method annotated with @AfterClass was run last, after all the other methods in the class. The test initializer method annotated with @Before was run before each test method.

The controls in the left side of the Test Results window enable you to easily run the test again. You can use the filter to toggle between displaying all test results or only the failed tests. The arrows enable you to skip to the next failure or the previous failure.

When you right-click a test result in the Test Results window, the popup menu enables you to choose to go to the test's source, run the test again or debug the test.

The next step after creating your unit test classes is to create test suites. See [Creating JUnit 4 Test Suites](https://netbeans.org/kb/docs/java/junit-intro.html#Exercise_42) to see how to run specified tests as a group so you do not have to run each test individually.

**Creating Test Suites**

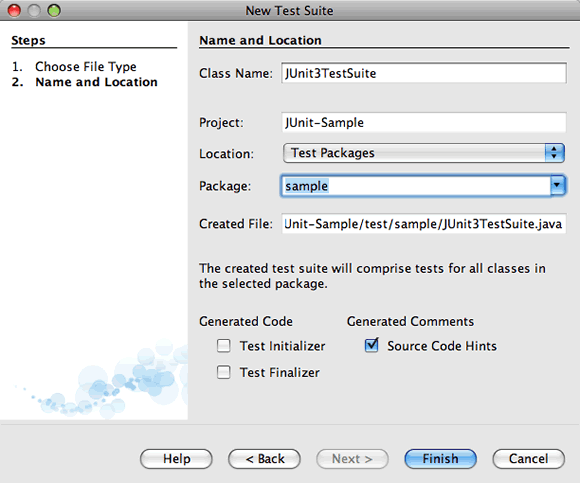
When creating tests for a project you will generally end up with many test classes. While you can run test classes individually or run all the tests in a project, in many cases you will want to run a subset of the tests or run tests in a specific order. You can do this by creating one or more test suites. For example, you can create test suites that test specific aspects of your code or specific conditions.

A test suite is basically a class with a method that invokes the specified test cases, such as specific test classes, test methods in test classes and other test suites. A test suite can be included as part of a test class but best practices recommends creating individual test suite classes.

You can create JUnit 3 and JUnit 4 test suites for your project manually or the IDE can generate the suites for you. When you use the IDE to generate a test suite, by default the IDE generates code to invoke all the test classes in the same package as the test suite. After the test suite is created you can modify the class to specify the tests you want to run as part of that suite.

**Creating JUnit 3 Test Suites**

If you selected JUnit 3 as the version for your tests, the IDE can generate JUnit 3 test suites based on the test classes in the test package. In JUnit 3 you specify the test classes to include in the test suite by creating an instance of TestSuite and using the addTest method for each test.

1. Right-click the **JUnit-Sample** project node in the Projects window and choose New > Other to open the New File wizard.
2. Select Test Suite in the Unit Tests category. Click Next.
3. Type **JUnit3TestSuite** for the Class Name.
4. Select the sample package to create the test suite in the sample folder in the test packages folder.
5. Deselect Test Initializer and Test Finalizer. Click Finish.  
   

When you click Finish, the IDE creates the test suite class in the sample package and opens the class in the editor. The test suite will contain the following code.

public class JUnit3TestSuite extends TestCase {

public JUnit3TestSuite(String testName) {

super(testName);

}

public static Test suite() {

TestSuite suite = new TestSuite("JUnit3TestSuite");

return suite;

}

}

1. Modify the suite() method to add the test classes that will be run as part of the suite.
2. public JUnit3TestSuite(String testName) {
3. super(testName);
4. }
5. public static Test suite() {
6. TestSuite suite = new TestSuite("JUnit3TestSuite");
7. **suite.addTest(new TestSuite(sample.VectorsJUnit3Test.class));**
8. **suite.addTest(new TestSuite(sample.UtilsJUnit3Test.class));**
9. return suite;

}

1. Save your changes.

**Creating JUnit 4 Test Suites**

If you selected JUnit 4 for the version of your tests, the IDE can generate JUnit 4 test suites. JUnit 4 is back-compatible so you can run JUnit 4 test suites that contain JUnit 4 and JUnit 3 tests. In JUnit 4 test suites you specify the test classes to include as values of the @Suite annotation.

**Note.** To run JUnit 3 test suites as part of a JUnit 4 test suite requires JUnit 4.4 or higher.

1. Right-click the project node in the Projects window and choose New > Other to open the New File wizard.
2. Select Test Suite in the Unit Tests category. Click Next.
3. Type **JUnit4TestSuite** for the file name.
4. Select the sample package to create the test suite in the sample folder in the test packages folder.
5. Deselect Test Initializer and Test Finalizer. Click Finish.

When you click Finish, the IDE creates the test suite class in the sample package and opens the class in the editor. The test suite contains code similar to the following.

@RunWith(Suite.class)

@Suite.SuiteClasses(value={UtilsJUnit4Test.class, VectorsJUnit4Test.class})

public class JUnit4TestSuite {

}

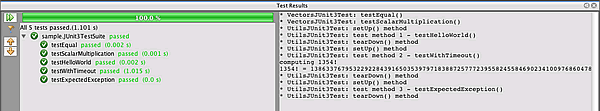
When you run the test suite the IDE will run the test classes in the order that they are listed.

**Running Test Suites**

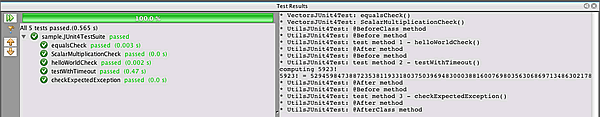
You run a test suite the same way you run any individual test class.

1. Expand the Test Packages node in the Projects window.
2. Right-click the test suite class and choose Test File.

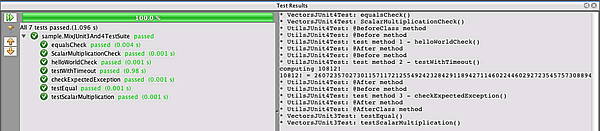
When you run the test suite the IDE runs the tests included in the suite in the order they are listed. The results are displayed in the JUnit Test Results window.

[](https://netbeans.org/images_www/articles/72/java/junit/junit3-suite-results.png)

In this image (click the image to see a larger image) you can see the test results for a JUnit 3 test suite. The test suite ran the UtilsJUnit3Test and VectorsJUnit3Test test classes as a single test and displayed the test results in the left pane as the results of a single test. The output in the right pane is the same as when you run the test individually.

[](https://netbeans.org/images_www/articles/72/java/junit/junit4-suite-results.png)

In this image (click the image to see a larger image) you can see the test results for a JUnit 4 test suite. The test suite ran the UtilsJUnit4Test and VectorsJUnit4Test test classes as a single test and displayed the test results in the left pane as the results of a single test. The output in the right pane is the same as when you run the test individually.

[](https://netbeans.org/images_www/articles/72/java/junit/junitmix3and4-suite-results.png)

In this image (click the image to see a larger image) you can see the test results for a mixed test suite. This test suite includes the JUnit 4 test suite and one of the JUnit 3 test classes. The test suite ran the UtilsJUnit3Test.java and JUnit4TestSuite.java test classes as a single test and displayed the test results in the left pane as the results of a single test. The output in the right pane is the same as running the test individually.

**Summary**

This tutorial was a basic introduction to creating JUnit unit tests and test suites in NetBeans IDE. The IDE supports JUnit 3 and JUnit 4, and this document demonstrated some of the changes introduced in JUnit 4 that are designed to make creating and running tests simpler.

As demonstrated in this tutorial, one of the main improvements in JUnit 4 is support for annotations. In JUnit 4 you can now use annotations to do the following:

* Identify a test using the @Test annotation instead of naming convention
* Identify setUp and tearDown methods with @Before and @After annotations
* Identify setUp and tearDown methods that apply to the entire test class. Methods annotated with @BeforeClass are run only once, before any test methods in the class are run. Methods annotated with @AfterClass are also run only once, after all the test methods have finished.
* Identify expected exceptions
* Identify tests that should be skipped using the @Ignore annotation
* Specify a timeout parameter for a test

For more information about using JUnit and other changes introduced in JUnit 4, see the following resources:

* [JUnit group at Yahoo groups](http://tech.groups.yahoo.com/group/junit/)
* [www.junit.org](http://www.junit.org)

Testing code often helps ensure that small changes made in the code do not break the application. Automated testing tools like JUnit streamline the process of testing and frequent testing can help catch coding errors early.