**JUnit has a graphical user interface (GUI), making it possible to write and test source code quickly and easily**. JUnit allows the developer to incrementally build test suites to measure progress and detect unintended side effects. Tests can be run continuously. Results are provided immediately.

**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.

This tutorial explains the use of JUnit in your project unit testing, while working with Java. After completing this tutorial you will gain sufficient knowledge in using JUnit testing framework from where you can take yourself to next levels.

* Name tests ***class-name*Test**. Such classes extend *TestCase* which is how JUnit is able to help in the testing effort.
* Name method tests (that test each method to be tested) **test*method-name***.
* In each *TestCase* (basic JUnit per-class testing class), override the *setUp* and *tearDown* methods in *TestClass* (see example below).

* The *setUp* method is run prior to every method test. Method tests may not usually depend on output or state created by another method test (and they should not try to). However, there is probably nothing wrong with them calling each other to achieve the effect.

* Add an *assert-* (usually) to each test case (see below).
* Keep tests in the same package as the classes they test.
* To avoid keeping the test code in the same physical directory as the implementations, create a test project package directory structure that mirrors the application structure.

* Gather great subsets of them into test suites named ***functionality*TestSuite**.
* Create a *TestSuite* in each package to test the classes there and another at each successively higher level to test more and more of the totality of the application.

* Use *addTestSuite* to collect individual class tests into a suite.
* Use *addTest* and not *addTestSuite* to add successively higher test suites (therefore: suites of suites) to JUnit tests in an application.

Setting up JUnit tests in Eclipse...

Here's how to wire up a JUnit library to your Eclipse project, it's pretty simple to do:

1. Right-click your project.
2. Choose Build Path -> Configure Build Path...
3. Click the Libraries tab.
4. Click Add Library...
5. Select JUnit.
6. Click Next.
7. Select version and click Finish.

At your project root, create a new folder named test.

1. Right-click it, choose Build Path, then Use as Source Folder.
2. Create a new package matching the package name in your project/*src* folder whose class(es) you want to test.

To Create a new JUnit test class:

1. Right-click the new test/package and choose JUnit Test Case (probably only available if you added the JUnit library in the steps above).
2. Follow the wizard as should be obvious.

Writing JUnit test code...

Is twofold: a) writing test cases and b) writing a test suite.

**Automatic test generation**

If using Eclipse, a test case for any class may be generated automatically with only the need to fill out stubs. The stubs are initially set up to generate failures, which concords with test-driven development (TDD) theory (everything should fail initially and implementation consists of eliminating the failures).

To set up a test case, locate the folder/package dominating the location the test source code will be place and right-click. Choose New->JUnit Test Case. Enter the name of the package (if it's wrong), the name of the test case (as the name of the class to test followed by “Test”), then click setUp() and tearDown() unless you don't want these for some reason, and last, the name of the class you're testing. Then, click next.

At this point, you're offered automatic code generation of those methods you select. Click Finish. You also have the option to create final method stubs and to create Eclipse tasks to remind you to finish implementing the test methods that are about to be generated.

**Writing the test methods...**

The test cases are written thus:

package *test-location*;

import junit.framework.TestCase;

public class ***class-name*Test** extends TestCase

{

protected void **setUp**()

{

// Set up data and objects required for use in any test case. Usually,

// these would be instance variables.

}

protected void **tearDown**()

{

// Release objects set up above.

}

public void **test*method-name***() <------ this is a test case

{

assertEquals( *expected*, *actual* );

}

}

Let's make a useful if bogus instantiation of the naming concepts above in order for later, *TestSuite* sample code to be of any use.

package com.myapp.utils;

import junit.framework.TestCase;

public class **FunAndGamesTest** extends TestCase

{

protected void **setUp**()

{

// Set up data and objects required for use in any test case. Usually,

// these would be instance variables.

}

protected void **tearDown**()

{

// Release objects set up above.

}

public void **testRobBank**()

{

assertEquals( *expected*, *actual* );

}

public void **testLivingTheVidaLoca**()

{

assertEquals( *expected*, *actual* );

}

}

The most important package in JUnit is **junit.framework**, which contains all the core classes. Some of the important classes are as follows −

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Class Name** | **Functionality** |
| 1 | Assert | A set of assert methods. |
| 2 | TestCase | A test case defines the fixture to run multiple tests. |
| 3 | TestResult | A TestResult collects the results of executing a test case. |
| 4 | TestSuite | A TestSuite is a composite of tests. |

## **Assert Class**

Following is the declaration for **org.junit.Assert** class −

public class Assert extends java.lang.Object

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

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **void assertEquals(boolean expected, boolean actual)**  Checks that two primitives/objects are equal. |
| 2 | **void assertFalse(boolean condition)**  Checks that a condition is false. |
| 3 | **void assertNotNull(Object object)**  Checks that an object isn't null. |
| 4 | **void assertNull(Object object)**  Checks that an object is null. |
| 5 | **void assertTrue(boolean condition)**  Checks that a condition is true. |
| 6 | **void fail()**  Fails a test with no message. |

Let's use some of the above-mentioned methods in an example. Create a java class file named TestJunit1.java in **C:\>JUNIT\_WORKSPACE**.

import org.junit.Test;

import static org.junit.Assert.\*;

public class TestJunit1 {

@Test

public void testAdd() {

//test data

int num = 5;

String temp = null;

String str = "Junit is working fine";

//check for equality

assertEquals("Junit is working fine", str);

//check for false condition

assertFalse(num > 6);

//check for not null value

assertNotNull(temp);

}

}

# **JUnit - Parameterized Test**

JUnit 4 has introduced a new feature called **parameterized tests**. Parameterized tests allow a developer to run the same test over and over again using different values. There are five steps that you need to follow to create a parameterized test.

* Annotate test class with @RunWith(Parameterized.class).
* Create a public static method annotated with @Parameters that returns a Collection of Objects (as Array) as test data set.
* Create a public constructor that takes in what is equivalent to one "row" of test data.
* Create an instance variable for each "column" of test data.
* Create your test case(s) using the instance variables as the source of the test data.

The test case will be invoked once for each row of data. Let us see parameterized tests in action.

# **JUnit - Extensions**

Following are the JUnit extensions −

* Cactus
* JWebUnit
* XMLUnit
* MockObject

## **Cactus**

Cactus is a simple test framework for unit testing server-side java code (Servlets, EJBs, Tag Libs, Filters). The intent of Cactus is to lower the cost of writing tests for server-side code. It uses JUnit and extends it. Cactus implements an in-container strategy that executes the tests inside a container.

Cactus ecosystem is made of several components −

* **Cactus Framework** is the heart of Cactus. It is the engine that provides the API to write Cactus tests.
* **Cactus Integration Modules** are front-ends and frameworks that provide easy ways of using the Cactus Framework (Ant scripts, Eclipse plugin, and Maven plugin).

## **JWebUnit**

JWebUnit is a Java-based testing framework for web applications. It wraps existing testing frameworks such as HtmlUnit and Selenium with a unified, simple testing interface to test the correctness of your web applications.

JWebUnit provides a high-level Java API for navigating a web application combined with a set of assertions to verify the application's correctness. This includes navigation via links, form entry and submission, validation of table contents, and other typical business web application features.

The simple navigation methods and ready-to-use assertions allow for more rapid test creation than using only JUnit or HtmlUnit. And if you want to switch from HtmlUnit to other plugins such as Selenium (available soon), there is no need to rewrite your tests.

Here is a sample code.

import junit.framework.TestCase;

import net.sourceforge.jwebunit.WebTester;

public class ExampleWebTestCase extends TestCase {

private WebTester tester;

public ExampleWebTestCase(String name) {

super(name);

tester = new WebTester();

}

//set base url

public void setUp() throws Exception {

getTestContext().setBaseUrl("http://myserver:8080/myapp");

}

// test base info

@Test

public void testInfoPage() {

beginAt("/info.html");

}

}

## **XMLUnit**

XMLUnit provides a single JUnit extension class, XMLTestCase, and a set of supporting classes that allow assertions to be made about −

* The differences between two pieces of XML (via Diff and DetailedDiff classes).
* The validity of a piece of XML (via Validator class).
* The outcome of transforming a piece of XML using XSLT (via Transform class).
* The evaluation of an XPath expression on a piece of XML (via classes implementing the XpathEngine interface).
* Individual nodes in a piece of XML that are exposed by DOM Traversal (via NodeTest class).

Let us assume we have two pieces of XML that we wish to compare and assert that they are equal. We could write a simple test class like this −

import org.custommonkey.xmlunit.XMLTestCase;

public class MyXMLTestCase extends XMLTestCase {

// this test method compare two pieces of the XML

@Test

public void testForXMLEquality() throws Exception {

String myControlXML = "<msg><uuid>0x00435A8C</uuid></msg>";

String myTestXML = "<msg><localId>2376</localId></msg>";

assertXMLEqual("Comparing test xml to control xml", myControlXML, myTestXML);

}

}

## **MockObject**

In a unit test, mock objects can simulate the behavior of complex, real (non-mock) objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test.

The common coding style for testing with mock objects is to −

* Create instances of mock objects.
* Set state and expectations in the mock objects.
* Invoke domain code with mock objects as parameters.
* Verify consistency in the mock objects.

Given below is an example of MockObject using Jmock.

import org.jmock.Mockery;

import org.jmock.Expectations;

class PubTest extends TestCase {

Mockery context = new Mockery();

public void testSubReceivesMessage() {

// set up

final Sub sub = context.mock(Sub.class);

Pub pub = new Pub();

pub.add(sub);

final String message = "message";

// expectations

context.checking(new Expectations() {

oneOf (sub).receive(message);

});

// execute

pub.publish(message);

// verify

context.assertIsSatisfied();

}

}