**Report on**

**JUnit**

**Content**

|  |  |
| --- | --- |
| Topic | Page Number |
|  |  |
| Introduction | 3 |
| JUnit | 4 |
| Annotations in JUnit | 5 |
| Asserts | 5 |
| Test exception | 6 |
| Test Fixtures | 6 |
| Conclusion | 7 |

**Introduction**

In SDLC, testing plays a vital role. When one tests a program one adds value to the program, in turn raising the quality and reliability of the program. When we say “reliable”, it implies finding and removing errors. Hence one should not test a program to show that it works, but to show that program does not work. Testing cannot guarantee against software problems or even failures but it can minimize the risks of faults developing once the software is put to use. Typically when testing one should start with assumptions that the program contains errors and the test the program to find as many errors as possible. Testing is a costly activity. A test which does not find an error is a waste of time and money. “A test case that finds an error is a valuable investment”.

The process of testing the individual subprograms, subroutines, or procedures to compare the function of the module to its specifications is called Unit testing. Unit testing is relatively inexpensive and an easy way to produce better code. Unit testing is done with the intent that a piece of code does what it is supposed to do.

Unit Testing is also called as Test Driven Development (TDD). A significant advantage of TDD is that it enables you to take small steps while writing software. Test Driven Development (TDD) requires developers to create automated unit tests that define code requirements before writing the code itself.

Unit testing helps developers find errors in code: When a developer starts writing unit tests, they can actually be surprised with how many errors are encountered in a small function that is written. It makes life of the developer easy. In case errors are “not found in time”, and are delayed till the end, then the entire module may fail. Testing helps you write better code: Unit testing can help developer during the initial phase of development. Unit testing makes designs better and drastically reduce the time required while debugging.

**JUnit:**

JUnit allows you to write tests faster while increasing quality and stability: Using JUnit, a developer spends less time debugging, and confidently makes changes to the code. With constant testing, any new functionality that is added can be verified for whether it is working or not. Hence the developer can be more positive about adding new features, because the developer now knows that it is less likely to fail. If a bug is detected while running tests, then the source code is fresh in your mind, so the bug is easily found. Also tests written in JUnit help you write code at a fast pace, and identify defects quickly. Writing tests builds the stability of the code and ensures that any changes that are made are working. As a result of the change, there is no effect on the software. JUnit is simple, elegant, and inexpensive: “Simplicity” is the keyword while writing a test.

Developers should not find it time consuming or difficult to write tests. With JUnit, the TDD can be followed very easily. It is simple and easy to put JUnit in practice. Developers can incrementally write tests as they increment their code. JUnit tests are such that they can be executed easily and frequently and it does not disturb the development process. This framework offers a cheap way of testing since it is an open source and freely downloadable ware.

JUnit tests check their own result and provide feedback immediately: Manual Unit testing is a tedious task and obviously it is time consuming to compare the expected and the actual result. As a result, developers tend to do away with Unit testing. JUnit tests can be run easily and they check their own results. The developer immediately gets a feedback if the tests have passed or failed. Hence any manual intervention is not required while executing the tests.

JUnit tests can be put together in a hierarchy of test suites: JUnit tests can be organized into test suites containing test cases and even other test suites. The composite behavior of JUnit tests allows you to assemble collections of tests and automatically regression test the entire test suite in one go. You can also run the tests for any layer within the test suite hierarchy.

JUnit tests are written in Java: Testing Java software using Java tests forms a “seamless bond” between the test and the code under test. The tests become an extension to the overall software and code can be refactored from the tests into the software under test. The Java compiler helps the testing process by performing static syntax checking of the unit tests and ensuring that the software interface contracts are being obeyed.

Annotations in JUnit are:

**@Test:**  It is used to signify a method as a test method. There are some options which can be mentioned with this annotation.

For example: @Test (timeout=100) fails if the test takes longer than 100 milliseconds for execution. This can be used to test infinite loops. Earlier you would have to start every method with ‘test’ which is now not required. The method can be named whatsoever. You have to simply prefix it with @Test annotation.

**@Before:** It is used to carry some task before each test is run. This can be used for initialization required before the test.

**@After:**  It is used to perform cleanup after each test is executed.

**@BeforeClass:** It will execute the method before the start of the tests. This can be used for initialization of intensive resources like database connection.

**@AfterClass:** It will execute the method after all tests have finished. Cleanup activities are carried out.

**@Ignore:** It will ignore the test method. This can be useful when the base code has been changed but the test is yet to be revised.

**Asserts:**

The org.junit.Assert class provides a set of useful assertions methods.

Some assertion methods are as follows:

Fail ([String]): It signals the failure of a test. This method has two formats. If the String argument is not provided, then no message is displayed. Else the String argument message is displayed.

assertTrue(boolean) : It asserts if the condition is true. Similarly assertFalse(boolean) asserts if the condition is false.

assertEquals([String message],expected,actual) : It asserts whether the two objects passed as arguments are equal. This method can accept any kind of values for comparison like double long, etc..

assertNull([message],object) : It asserts that an object is null.

assertNotNull([message],object) : It asserts that an object is not null.

assertSame([String],expected,actual) : It asserts that two objects refer to the same object and assertNotSame([String],expected,actual) asserts that two objects do not refer to the same object.

assertThat(String, T actual, Matcher <T> matcher) : It asserts that “actual” satisfies the condition specified by the “matcher”.

**Test exception:**

Example of Testing with JUnit – Testing Exceptions:

How do you verify that code throws exceptions as expected? Verifying that code completes normally is important, but making sure the code behaves as expected in exceptional situations is vital too.

For example:

new ArrayList<Object>().get(0);

This code should throw an IndexOutOfBoundsException. The @Test annotation has an optional parameter "expected" that takes as values subclasses of Throwable. If we wanted to verify that ArrayList throws the correct exception, we would write:

@Test(expected = IndexOutOfBoundsException.class)

public void empty() {

new ArrayList<Object>().get(0);

}

The expected parameter should be used with care. The above test will pass if any code in the method throws IndexOutOfBoundsException.

**Test Fixtures:**

Test Fixtures includes two types of annotations used by the method while executing:

1.@BeforeClass

2.@AfterClass

Using @BeforeClass and @AfterClass:

Sometimes the requirement can be wherein the initialization code is run, and then several tests are executed. In this scenario, we need to use @BeforeClass and @AfterClass annotated methods. This can also be termed as “one time setup and teardown”.

@BeforeClass: The method annotated with this will be executed once before the tests are run. It means if you have just one test or ten tests, then this will run one time before the very first test executed. In case of inheritance, the base class @BeforeClass method will execute once.

@After class: The method annotated with this will run once after all the tests have finished execution. In case of inheritance, the @AfterClass in the derived class will execute first and then the method from base class will be executed.

Unlike @Before and @After, only one set of @BeforeClass and @AfterClass annotated methods are allowed.

**Conclusion**

* JUnit framework provides a standard way of writing test cases
* Simple framework for writing automated, self-verifying tests in Java
* Support for test assertions
* Immediate test reporting