Junit Testing Framework

WHAT IS TESTING?

- Testing is the process of evaluating an application to check whether it is satisfying the desired requirements or not.

- Testing enables us to find out if there are errors, gaps and missing requirements in the application.

BENEFITS OF TESTING

- It improves software design and make it easy to understand.
- It finds bugs and errors during various stages of software development.
- It reduces possibility of failure of an application to ZERO. Failures in future can be very costly.
- It increases performance of application.
- It ensures the quality of the application and simultaneously make sure that application is meeting its goals.
- It enhances reliability of application.
- It makes sure that your application doesn’t negatively affect interacting systems.

TYPES OF TESTING

- Unit Testing
- Integration Testing
- Functional Testing
- Performance Testing

UNIT TESTING

- *Unit testing* is a process in which the smallest testable parts of an application, called units, are individually and independently checked for proper functioning.

- Unit test
  - is a piece of code that invokes a unit of a work in the application for testing.
  - ensures that code should do its desired task.
  - targets only small unit of code, a method or a class

INTEGRATION TESTING

- In Integration Testing, individual software modules are integrated logically and tested as a group, after completion of unit testing.

- The purpose of integration testing is to verify the functional, performance, and reliability between the modules that are integrated.

- Integration Testing Strategies:
  - Big-Bang Integration,
  - Top Down Integration,
  - Bottom Up Integration, and
  - Hybrid Integration.

FUNCTIONAL TESTING

- In functional testing basically the testing of the functions of component or system is done.

- It refers to activities that verify a specific action or function of the code.

- Functional test tends to answer the questions like “can the user do this” or “does this particular feature work”.
  - This is typically described in a requirements specification or in a functional specification.

- Function Testing strategies can be of two types:
  - Black box Testing
  - White Box Testing

PERFORMANCE TESTING

- It is a testing technique to determine the speed, effectiveness, reliability of an application during various workload conditions.

- This type of testing is done to measure the quality attributes of the application, i.e. reliability, scalability, and resource utilization.

- It also verifies that an application meets the specifications.

WHEN TO START TESTING?

- It is better, to start testing at the early stage of software development. In SDLC, testing can be started from Requirement phase itself.
- It also depends on the development model that is being used to developing application.
- Testing can be done in different ways depending on the phase. Following examples can be considered as testing:
  - In requirement phase – analysis and verification of requirements
  - In design phase – reviewing the design in the design phase
  - After code completion – tests performed by developer

WHEN TO STOP TESTING?

- Testing is on-going process, it is difficult to tell that an application is 100% tested.
- In SDLC, Testing is started at requirement phase, can be done till deployment phase.
- It is very difficult to decide when to stop testing. However there are certain parameter to decide to stop testing.
  - Testing deadlines
  - Completion of test case execution
  - Completion of functional and code coverage to a certain point
  - Bug rate falls below a certain level
  - No high-priority bugs are identified
  - Management decision

VERIFICATION VS VALIDATION

VERIFICATION

- It is the process of evaluating system in the development phase to find out whether they meet the specified requirements.
- It takes place first and includes the checking for documentation, code etc.
- Reviews, meetings and inspections are involved.
- It is basically manual checking the documents and files like requirement specifications etc.
- It is done by developers.

VALIDATION

- It is the process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements.
- It occurs after verification and involves the checking of the overall product.
- Testing techniques involved are black box testing, white box testing, gray box testing etc.
- It is basically checking of developed program based on the requirement specifications documents.
- It is done by testers.

TESTING FRAMEWORKS

- A testing framework is a set of assumptions, concepts, tools and practices that provides support to testing.

- Testing framework provides an execution environment for software testing.

NEED OF TESTING FRAMEWORKS

- Projects implement unique strategies. Time needed for the tester to become productive in the new environment takes long.
- A testing framework that is application independent and has the capability to expand with the requirements of each application.
- An organized test framework helps in avoiding duplication of test cases automated across the application.
- A test framework helps teams organize their test suites and in turn improves the efficiency of testing.
- Each class must be tested when it is developed and needs a regression test.
- Regression tests need to have standard interfaces. Thus, we can build the regression test when building the class and have a better, more stable product for less work.

TESTING FRAMEWORKS FOR JAVA

- There are many testing frameworks available in java. Some of them are :
  - JUnit
  - TestNG

INTRODUCTION TO JUNIT TESTING FRAMEWORK

- JUnit
  - is a Regression Testing Framework to implement unit testing in Java.
  - is simple to use.
  - writes repeatable tests.
  - is open source framework
  - belongs to a family of unit testing frameworks “xUnit”
  - was originally written by Erich Gamma and Kent Beck.

FEATURES OF JUNIT

- Junit provides :
  - test runners to run tests.
  - test suites to organize test cases.
  - annotations to identify the test methods
  - assertions for testing expected results

- JUnit is used to test:
  - an entire object
  - part of an object – a method or some interacting methods
  - interaction between several objects

UNIT TEST CASES IN JUNIT

- A Unit Test Case is a part of code which ensures that the another part of code (method) works as per the expectations.

- To achieve those expected results quickly, test framework is needed. JUnit is perfect unit test framework for java programming language.

- A formal written unit test case is characterized by a known input and by 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.

- Each requirement must have at least two test cases : one positive test and one negative

JUNIT NAMIMG CONVENTIONS

- There are some important naming conventions for Junit. They are :
  - Add the “Test” suffix with test class name
  - Use the word “Should” in the test method name
  - Test name should be able to convey its implementation

JUNIT ANNOTATIONS

Annotation	Description
@Test public void method()	The @Test annotation identifies a method as a test method
@Test(expected = Exception.class)	Fails if method doesn’t throw the mentioned exception
@Test(timeout = 100)	Test fails if it takes more than 100 milliseconds
@Before public void method()	This method will run before each test method
@After public void method()	This method will run after each test method
@BeforeClass public static void method()	This method will be called once per test class, before execution of all the test methods.
@AfterClass public static void method()	This method will be called once per test class, after execution of all the test methods.
@Ignore @Test public void method()	Method annotated with @Test that is also annotated with @Ignore will not be executed as test

JUNIT ASSERTIONS

- All the assertions are available in the Assert class of java.lang package.

- Assert class provides assertion methods for writing tests.

- Junit provides overloaded assertion methods for all primitive types, arrays and Objects.

JUNIT ANNOTATIONS

Assertion Method	Description
void assertEquals(boolean expected, boolean actual)	Checks that two objects are equal
Void assertTrue(Boolean expected, Boolean actual)	Checks that a condition is true
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 assertSame(boolean condition)	Checks if two object references point to the same object
void assertNotSame(boolean condition)	Checks if two object references not point to the same object
void assertArrayEquals(expectedArray, resultArray)	Tests whether two arrays are equal

TESTING EXCEPTIONS

- It is easy to trace the Exception handling of code in JUnit.
- Code can be tested, whether code throws desired exception or not.
- With @Test annotation, expected parameter is used .

```
@Test(expected=ArithmeticException.class)
public void division(){
            int i = 1/0;
}
```

TEST FIXTURE

- A *test fixture* is a fixed state in code which is tested used as input for a test. Another way to describe this is a test precondition.
- For example,
  - Loading a database with a specific, known set of data
  - Copying a specific known set of files
  - Preparation of input data and setup/creation of fake or mock objects

- In other word, creating a test fixture is to create a set of objects initialized to certain states.

JUNIT TEST FIXTURE

- There are four fixture annotations:
  - Two for class-level fixtures –
    - @BeforeClass and
    - @AfterClass
  - Two for method-level –
    - @Before and
    - @After

JUNIT TEST FIXTURE : EXAMPLE (1 of 2)

```
@BeforeClass
public static void setUpClass() {
        System.out.println("@BeforeClass setUpClass");
        myExpensiveManagedResource = new ExpensiveManagedResource();
}

@AfterClass
public static void tearDownClass() throws IOException {
        System.out.println("@AfterClass tearDownClass");
        myExpensiveManagedResource.close();
        myExpensiveManagedResource = null;
}
```

JUNIT TEST FIXTURE : EXAMPLE (2 of 2)

```
@Before
public void setUp() {
       this.println("@Before setUp");
       this.myManagedResource = new ManagedResource();
}

@After
public void tearDown() throws IOException {
       this.println("@After tearDown");
       this.myManagedResource.close();
       this.myManagedResource = null;
}
```

JUNIT TEST CASE : EXAMPLE (1/2)

- Assume we have Counter class for testing

  public class CounterTest extends junit.framework.TestCase {

  This is the unit test for the Counter class

  public CounterTest() { } //Default constructor

  protected void setUp()

  Test *fixture* creates and initializes instance variables, etc.

  protected void tearDown()

  Releases any system resources used by the test fixture

  public void testIncrement(), public void testDecrement()

  These methods contain tests for the Counter methods increment(), decrement(), etc.

JUNIT TEST CASE : EXAMPLE (2/2)

```
 public class CounterTest extends junit.framework.TestCase {
    Counter counter1;
    public CounterTest() { }   // default constructor

    protected void setUp() {   // creates a (simple) test fixture
        counter1 = new Counter();
    }

    public void testIncrement() {
        assertTrue(counter1.increment() == 1);
        assertTrue(counter1.increment() == 2);
     }

    public void testDecrement() {
        assertTrue(counter1.decrement() == -1);
    }
}
```

PARAMETERIZED TEST

- Junit 4 has included a new feature of parameterized test. This test allows user to run same test repeatedly using different values.
- 5 Steps to create parameterized test
  - Use annotation @RunWith(Parameterized.class) with test class
  - Write a public static method with @Parameters annotation
  - Write a public constructor
  - Create an instance variable that takes each column of test data
  - Create your test case using the instance variables as the source of data

PARAMETERIZED TEST : EXAMPLE (1/2)

```
 public class Calculate {
    public int sum(int var1, int var2) {
       System.out.println("Adding values: " + var1 + " + " + var2);
       return var1 + var2;
       }
      }
```

```
 @RunWith(Parameterized.class)
 public class ParameterizedTest {
    private int expected;
    private int first;
    private int second;
 public ParameterizedTest(int expectedResult, int firstNumber, int
secondNumber) {
 this.expected = expectedResult;
 this.first = firstNumber;
 this.second = secondNumber; }
```

PARAMETERIZED TEST : EXAMPLE (2/2)

```
@Parameters
public static Collection addedNumbers() {
   return Arrays.asList(new Integer[][] { { 3, 1, 2 }, { 5, 2, 3 }, { 7,
   3, 4 }, { 9, 4, 5 }, });
}
@Test
public void sum() {
   Calculate add = new Calculate();
   System.out.println("Addition with parameters : " + first + " and " +
   second);
   assertEquals(expected, add.sum(first, second));
}
}
```

RULES IN JUNIT

- Rules are used to add additional functionality which applies to all tests within a test class, but in a more generic way.

- Rules allow very flexible addition or redefinition of the behaviour of each test method in a test class.

- Testers can reuse or extend existing rules, or write their own.

- @Rule annotation is used to mark public fields of a test class.

JUNIT RULE EXAMPLE

```
@Rule
public TestName name = new TestName();

@Test
public void testA() {
        System.out.println(name.getMethodName());
        assertEquals("testA", name.getMethodName());
}
@Test
public void testB() {
        System.out.println(name.getMethodName());
        assertEquals("testB", name.getMethodName());
}
```