infogain

Unit Testing Using JUnit

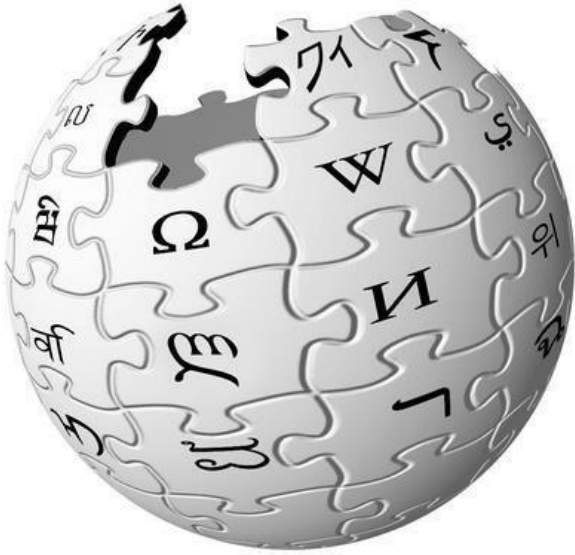
Objective

www.infogain.com

- Introduction to Unit Testing
- Difference between JUnit 3.x and JUnit 4.x
- Introduction to JUnit 4.x
- Exploring Junit Test Runner
- Exploring JUnit 4.x annotations
- Verifying test conditions with assertion
- Working with the @RunWith annotation and Ignoring a test
- Exploring the test suite
- Working with timeouts
- Exploring JUnit matchers
- Exploring JUnit parameterized test
- Exploring Junit rules

Unit Test : A Definition

A unit test is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward.
If the assumptions turn out to be wrong, the unit test has failed.
Unit testing is usually performed by the developer.



Note : Unit testing will be performed against a system under test (SUT).

Unit Test : An Introduction

- A procedure to validate individual units of source code
- Example: A procedure or method
- Validating each individual piece reduces errors when integrating the pieces together later
- A unit test is a test related to a single responsibility of a single class, often referred to as the System Under Test (SUT).
- A test is an assessment of our knowledge, a proof of concept, or an examination of data.
- Unit testing code means validation or performing the sanity check of code.
- Sanity check is a basic test to quickly evaluate whether the result of a calculation can possibly be true.

Difference between JUnit 3.x and JUnit 4.x

Feature	JUnit 3.x	JUnit 4.x
test annotation	testXXX pattern	@Test
run before the first test method in the current class is invoked	None	@BeforeClass
run after all the test methods in the current class have been run	None	@AfterClass
run before each test method	override setUp()	@Before
run after each test method	override tearDown()	@After
ignore test	Comment out or remove code	@ignore
expected exception	catch exception assert success	@Test(expected = ArithmeticException.class)
Timeout	None	@Test(timeout = 1000)

How to Write a Unit Test ?

- Using a Framework
- Without using a Framework

- **Benefits of using Unit-testing frameworks .**
  - Write tests more quickly with a set of known APIs,
  - Execute those tests automatically,
  - Review the results of those tests easily.

List of Unit Testing Framework Available for Java

- **SpryTest**
- **Jtest**
- **JUnit**
- **TestNG**

What is JUnit?

- A light-weight open-source testing framework for Java developed in Java by Kent Beck and Erich Gamma
- Classes and source code can be downloaded from www.junit.org
- Popular with Java developers for automated unit-testing
- Corresponding versions available for C, C++, etc.
- Facilitates Test-Driven Development

What is a test?

- As per JUnit working, a test is a “matching” of an expected value object with a “result” or “computed” value object.
- A test “passes” if the two are equal; it “fails” if the two are unequal.
- The two objects being compared must be of the same type. They can be of type int, String, or a user-defined object. A user-defined object must have appropriate equals() and toString() methods defined; these will form the basis for JUnit comparison and reporting.
- A test can be an expression being true or false

Junit Test Runner

- It is a component that instantiates our test classes.
- It runs test methods found from our test classes.
- Reports the test results.

- Rules and Limitations
- All test runners must extends the abstract org.junit.runner.Runner class
- A test class have only one test runner.
- The default test runner is called the BlockJunit4Classrunner.

Exploring Junit 4.x Annotations

- JUnit 4 is an annotation-based framework
- The @Test annotation represents a test. Any public method can be annotated with the @Test annotation to make it a test method.
- @Before annotate any public void method of any name, then that method gets executed before every test execution.
- @After gets executed after each test method execution
- @BeforeClass and @AfterClass annotations can be used with any public static void methods.
- @BeforeClass annotation is executed before the first test and the @AfterClass annotation is executed after the last test.

Verifying test conditions with Assertion

- Assertion is a tool (a predicate) used to verify a programming assumption (expectation) with an actual outcome of a program implementation
- assertTrue(condition) or assertTrue(failure message, condition)
- assertFalse(condition) or assertFalse(failure message, condition)
- assertNull:
- assertNotNull:
- assertEquals(string message, object expected, object actual), or assertEquals(object expected, object actual), or

Verifying test conditions with Assertion (contd..)

- assertEquals(primitive expected, primitive actual): T
- assertSame(object expected, object actual): This supports only objects and checks the object reference using the == operator.
- assertNotSame: This is just the opposite of assertSame. It fails when the two argument references are the same.
- Working with exception handling :
- @Test annotation takes the expected=<<Exception class name>>.class

Exploring the test suite

- To run multiple test cases, JUnit 4 provides Suite.class and the @SuiteClasses annotation.
- This annotation takes an array (comma separated) of test classes.
- Create a TestSuite class and annotate the class with
- @RunWith(Suite.class).
- This annotation will force Eclipse to use the suite runner.

```
@RunWith(Suite.class)
@SuiteClasses({ AssertTest.class, AssertThatTest.class, Assumption.class,
        IgnoreTest.class })
```

Working with JUnit 4++

- Ignoring a test :

```
@Test
@Ignore("This case is going to ignore")
public void when_today_is_holiday_then_stop_alarm() {

    Assert.assertNull(3);
}
```

- Executing tests in order :
- JUnit was designed to allow execution in a random order, but typically they are executed in a linear fashion and the order is not guaranteed.

```
@FixMethodOrder(MethodSorters.DEFAULT)
public class TestExecutionOrder {
@Test public void edit() throws Exception { System.out.println("edit executed"); }
@Test public void create() throws Exception { System.out.println("create executed");}
@Test public void remove() throws Exception { System.out.println("remove executed"); }
```

- We can assume some cases due that our test gets fail. org.junit.Assume

Asserting with assertThat : Using Matcher

- Joe Walnes created the assertThat(Object actual, Matcher matcher) method.
- General consensus is that assertThat is readable and more useful than assertEquals.
- public static void assertThat(Object actual, Matcher matcher)
- The Matcher methods use the builder pattern so that we can combine one or more matchers to build a composite matcher chain.

```
@Test
public void verify_Matcher() throws Exception {
    int age = 30;
    assertThat(age, equalTo(30));
    assertThat(age, is(30));
```

Using Matcher (Cont..)

- Working with compound value matchers : either, both, anyOf, allOf, and not

```
@Test
public void verify_multiple_values() throws Exception {
double marks = 100.00;
assertThat(marks, either(is(100.00)).or(is(90.9)));
assertThat(marks, both(not(99.99)).and(not(60.00)));
assertThat(marks, anyOf(is(100.00),is(1.00),is(55.00),
is(88.00),is(67.8)));
assertThat(marks, not(anyOf(is(0.00),is(200.00))));
assertThat(marks, not(allOf(is(1.00),is(100.00), is(30.00))));
}
```

- Working with collection matchers – hasItem and hasItems

```
@Test
public void verify_collection_values() throws Exception {
List<Double> salary =Arrays.asList(50.0, 200.0, 500.0);
assertThat(salary, hasItem(50.00));
assertThat(salary, hasItems(50.00, 200.00));
assertThat(salary, not(hasItem(1.00)));
}
```

Using Matcher (Cont..)

- Exploring string matchers – startsWith, endsWith, and containsString

```
assertThat(name, startsWith("John"));
assertThat(name, endsWith("Dale"));
assertThat(name, containsString("Jr"));
```

Creating parameterized tests

- Parameterized tests are used for multiple iterations over a single input to stress the object in test.
- The primary reason is to reduce the amount of test code.
- In TDD, the code is written to satisfy a failing test.
- We read about the @RunWith annotation in the preceding section. Parameterized is a special type of runner and can be used with the @RunWith annotation.
- Parameterized comes with two flavors: constructor and method.

```
@RunWith(Parameterized.class)
public class ParameterizedFactorialTest {
```

- The Parameterized runner needs a constructor to pass the collection of data. For each row in the collection, the 0th array element will be passed as the 1st constructor argument, the next index will be passed as 2nd argument, and so on.

Creating parameterized tests (cont..)

- Working with parameterized methods:
- We learned about the parameterized constructor; now we will run the parameterized test excluding the constructor.

```
@Parameters
  public static Collection<Object[]> factorialData() {
        return Arrays.asList(new Object[][] {
{ 0, 1}, { 1, 1 }, { 2, 2 }, { 3, 6 }, { 4, 24 }, { 5, 120 },{ 6, 720 }
          });}
```

Creating parameterized tests (cont..)

- If we run the test, it will fail as the reflection process won't find the matching constructor. JUnit provides an annotation to loop through the dataset and set the values to the class members. @Parameter(value=index) takes a value.

```
@Parameter(value=0)
public int number;
@Parameter(value=1)
public int expectedResult;
```

- The @Parameters annotation allows placeholders that are replaced at runtime, and we can use them. The following are the placeholders:
- {index}: This represents the current parameter index
- {0}, {1},…: This represents the first, second, so on parameter values

Working with JUnit rules

- Rules allow very flexible addition or redefinition of the behavior of each test method in a test class.
- We can use the inbuilt rules or define our custom rule .
- Playing with the timeout rule :

```
@Rule
public Timeout globalTimeout = new Timeout(30000,TimeUnit.MILLISECONDS);
```

- ExpectedException rule :
- The ExpectedException rule allows in-test specification of expected
- exception types and messages.

```
@Rule
public ExpectedException thrown= ExpectedException.none();
```

Working with JUnit rules

- TemporaryFolder rule :
- The TemporaryFolder rule allows the creation of files and folders that are guaranteed to be deleted when the test method finishes (whether it passes or fails).

```
@Rule
public TemporaryFolder folder = new TemporaryFolder();
```

- ErrorCollector rule :
- The ErrorCollector rule allows the execution of a test to continue after the first problem is found (for example, to collect all the incorrect rows in a table and report them all at once) as follows:

```
@Rule
public ErrorCollector collector = new ErrorCollector();
```

Working with JUnit rules

- Verifier rule :
- Verifier is a base class of ErrorCollector, which can otherwise turn passing tests into failing tests if a verification check fails .

```
@Rule
public TestRule rule = new Verifier() {
    protected void verify() {
    };
```

infogain

Thank You