infogain

Working with Junit 4.x

April 22, 2019

Objective

- Introduction to JUnit 4
- Using JUnit within Eclipse
- Working with Assertion and Working with exception handling
- Working with the @RunWith annotation and Ignoring a test
- Executing tests in order and Working with assumptions
- Exploring the test suite and Asserting with assertThat
- Working with matchers and Creating parameterized tests
- Working with timeouts
- Exploring JUnit theories
- Externalizing data using @ParametersSuppliedBy and ParameterSupplier
- Working with the ExpectedException rule
- Working with TestRules

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

"Never in the field of software development was so much owed by so many to so few lines of code"
- Martin Fowler, author of "UML Distilled" & "Refactoring", speaking about JUnit

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

Exploring annotations

JUnit 4 is an annotation-based framework

- The ***@Test*** annotation represents a test. Any public method can be annotated with the @Test annotation with @Test 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
- 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 @RunWith annotation

- When a class is annotated with @RunWith or the class extends a class annotated with @RunWith, JUnit will invoke the class that it references to run the tests on that class, instead of using the built-in runner.

- The @RunWith annotation is used to change the nature of the test class. It can be used to run a test as a parameterized test or even a Spring test, or it can be a Mockito runner to initialize the mock objects annotated with a @Mock annotation.

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

}
```

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

Exploring the test suite

- To run multiple test cases, JUnit 4 provides **Suite.class** and the **@Suite**. **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 })
```

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)));
}
```

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.
- The production code logic is built from a set of test cases and different input values
- 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 }
          });}
```

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

Exploring JUnit theories

- A theory is a kind of a JUnit test but different from the typical example-based JUnit tests, where we assert a specific data set and expect a specific outcome.
- JUnit theories are an alternative to JUnit's parameterized tests.
- Parameterized tests allow us to write flexible data-driven tests and separate data from the test methods.
- Theories are similar to parameterized tests—both allow us to specify the test data outside of the test case.

Exploring JUnit theories cont..

Parameterized tests are good but they have the following drawbacks:

- Parameters are declared as member variables. They pollute the test class and unnecessarily make the system complex.
- Parameters need to be passed to the single constructor or variables need to be annotated, simply making the class incomprehensible.
- Test data cannot be externalized.

Theory Annotations

- **@Theory**: Like @Test, this annotation identifies a theory test to run. The @Test annotation doesn't work with a theory runner.
- **@DataPoint:** This annotation identifies a single set of test data (similar to @Parameters), that is, either a static variable or a method.
- **@DataPoints:** This annotation identifies multiple sets of test data, generally an array.
- **@ParametersSuppliedBy**: This annotation provides the parameters to the test cases.
- **@Theories:** This annotation is a JUnit runner for the theory-based test cases and extends **org.junit.runners.BlockJUnit4ClassRunner.**
- **@ ParameterSupplier:** This is an abstract class that gives us the handle on the parameters that we can supply to the test case.

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();
```

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();
```

Working with JUnit rules

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();
```

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

April 22, 2019