JUnit
Testing Framework

Sang Shin
www.javapassion.com
"Learn with Passion!"

Topics

- What is and Why JUnit framework?
- How to write JUnit testing code?
- Fixtures
- Test suites
- Test runners (Only in JUnit 3.x)
- JUnit 4.x (Annotation-based)
- Best practice guidelines

What is & Why JUnit Testing Framework?

What is JUnit?

- JUnit is a regression testing framework
- Used by developers to implement unit tests in Java (de-facto standard)
- Integrated with Ant, Maven
  - Testing is performed as part of nightly build process: You can tell what has been broken
- Goal: Increase the quality of code.
- Part of XUnit family (HTTPUnit, Cactus for Server side testing, etc)

Why JUnit?

- Without JUnit, you will have to use println() to print out some result
  - No explicit concept of test passing or failure
  - No mechanism to collect results in a structured fashion
  - No replicability
- JUnit addresses all these issues

Key Goals of JUnit

- Easy to create tests
- Create tests that retain their value over time
- Leverage existing tests to write new ones (reusable)

What does JUnit provide?

- API for easily creating Java test cases
- Comprehensive assertion facilities
  - verify expected versus actual results
- Test runners for running tests (only in JUnit 3.x)
- Aggregation facility (test suites)
- Reporting

How to write JUnit Testing code?

How to write JUnit-based testing code (Minimum) – JUnit 3.x

- Include JUnit_<version>.jar in the classpath
- Define a subclass of **`TestCase`** class
- Define one or more public **`testXXX()`** methods in the subclass
  - Write assert methods inside the testXXX methods()
- Optionally define **`main()`** to run the TestCase in batch mode.

Rules of testXXX() methods

- Test methods has name pattern testXXX() - JUnit 3.x
- Test methods must have no arguments
- Test methods return nothing - void

Example 1: Very Simple Test (JUnit 3.x)

```
import junit.framework.TestCase;

public class SimpleTest extends TestCase {

    public SimpleTest(String name) {
        super(name);
    }
    // Test code
    public void testSomething() {
        System.out.println("About to call assertTrue() method...");
        assertTrue(4 == (2 * 2));
    }
    // You don't have to have main() method, You can use Test runner
    public static void main(String[] args){
        junit.textui.TestRunner.run(SimpleTest.class);
    }
}
```

Demo:

TestComputeClass
1048_javase_junit.zip

setUp() and tearDown() & Suite

- Optionally override the `setUp()` & `tearDown()` methods
  - Use `setUP()` for setting things up before each test
  - Use `tearDown()` for tearing things down after each test
- Optionally define a static `suite()` factory method
  - Create a TestSuite containing all the tests

Example 2: Using setUp() & tearDown()

```
// Define a subclass of TestCase
public class StringTest extends TestCase {

    // Create fixtures
    protected void setUp(){ /* run before */}
    protected void tearDown(){ /* after */ }

    // Add testing methods
    public void testSimpleAdd() {
      String s1 = new String("abcd");
      String s2 = new String("abcd");
      assertTrue("Strings equal",
                 s1.equals(s2));
     }

    // Could run the test in batch mode
    public static void main(String[] args){
     junit.textui.TestRunner.run (StringTest.class);
    }
```

Demo:

TestComputeClass_setUptearDown
1048_javase_junit.zip

Assert Statements

Assert statements

- JUnit Assertions are methods starting with assert
- Determines the success or failure of a test
- An assert is simply a comparison between an expected value and an actual value
- Two variants
  - assertXXX(...)
  - assertXXX(String message, ...) - the message is displayed when the assertXXX() fails

Assert statements

- Asserts that a condition is true
  - assertTrue(boolean condition)
  - assertTrue(String message, boolean condition)
- Asserts that a condition is false
  - assertFalse(boolean condition)
  - assertFalse(String message, boolean condition)

Assert statements

- Asserts expected.equals(actual) behavior
  - assertEquals(expected, actual)
  - assertEquals(String message, expected, actual)
- Asserts expected == actual behavior
  - assertSame(Object expected, Object actual)
  - assertSame(String message, Object expected, Object actual)

Assert statements

- Asserts object reference is null
  - assertNull(Object obj)
  - assertNull(String message, Object obj)
- Asserts object reference is not null
  - assertNotNull(Object obj)
  - assertNotNull(String message, Object obj)
- Forces a failure
  - fail()
  - fail(String message)

Fixtures

Fixtures

- *setup()* and *tearDown()* used to initialize and release common test data, which are called fixtures
- *setup()* is run before every test invocation & *tearDown()* is run after every test method

Test Suites

Test Suites (JUnit 3.x)

- Used to collect all the test cases
- Suites can contain testCases and testSuites
  - addTest(Test test) or addTestSuite(java.lang.Class testClass)

Example: Test Suites (JUnit 3.x)

```
public static void main (String [] args){
    junit.textui.TestRunner.run (suite ());
}

public static Test suite (){
   suite = new TestSuite  ("AllTests");
   suite.addTest
      (new TestSuite (AllTests.class));
   suite.addTest (StringTest.suite());
}

public void testAllTests () throws Exception{
     assertTrue (suite != null);
}
```

Example: Test Suites (JUnit 3.x)

```
// TestSuite implements Test interface
public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new TestSuite(FloatTest.class));
        suite.addTest(new TestSuite(BoolTest.class));
        return suite;
 }
```

Demo:

TestComputeClass_TestSuite
1048_javase_junit.zip

Test Runners

TestRunners (JUnit 3.x)

- Text
  - Lightweight, quick quiet
  - Run from command line

```
java StringTest
.......
Time: 0.05
Tests run: 7,  Failures: 0,  Errors: 0
```

Automating testing (Ant)

- JUnit Task

```
<target name="test" depends="compile-tests">
  <junit printsummary="yes" fork="yes">
       <classpath>
        <pathelement location="${build}" />
         <pathelement location="${build}/test" />
       </classpath>
       <formatter usefile="yes" type="plain" />
       <test name="AllTests" />
   </junit>
 </target>
```

Ant Batch mode

```
<target name="batchtest" depends="compile-tests">
    <junit printsummary="yes" fork="yes" haltonfailure="no">
    <classpath>
        <pathelement location="${build.dir}" />
        <pathelement location="${build.dir}/test" />
     </classpath>
    <formatter type="plain" usefile="yes"/>

      <batchtest fork="yes" todir="">
       <fileset dir="${test.dir}">
            <include name="**/*Test.java" />
       </fileset>
     </batchtest>

   </junit>
 </target>
```

JUnit 4.x

JUnit4 Improvements over JUnit3

- Test class no longer needs to extend *TestCase* class
- Test methods no longer needs to be named as *test<NameOfMethod>*
  - *@Test* annotation over any arbitrary named method
- setup and teardown methods no longer have to be named as *setUp()* and *tearDown()*
  - *@Before* and *@After* over any arbitrary named method
- One-time setup and teardown annotations
  - *@BeforeClass, @AfterClass*
- Timeout and Ignore annotations
  - @Test(timeout = 5000), @Ignore("Ignore this test for now")
- TestSuite annotations

Example: JUnit 3 based

```
// In JUnit 3, you have to extend TestCase class
public class SimpleInterestCalculatorJUnit38Tests extends TestCase {

    private InterestCalculator interestCalculator;

    protected void setUp() throws Exception {
        interestCalculator = new SimpleInterestCalculator();
        interestCalculator.setRate(0.05);
    }

    public void testCalculate() {
        double interest = interestCalculator.calculate(10000, 2);
        assertEquals(interest, 1000.0);
    }

    public void testIllegalCalculate() {
        try {
            interestCalculator.calculate(-10000, 2);
            fail("No exception on illegal argument");
        }
        catch (IllegalArgumentException e) {}
    }
}
```

Example: JUnit 4 based

```
public class SimpleInterestCalculatorJUnit4Tests {

    private InterestCalculator interestCalculator;

    @Before
    public void init() {
        interestCalculator = new SimpleInterestCalculator();
        interestCalculator.setRate(0.05);
    }

    @Test
    public void calculate() {
        double interest = interestCalculator.calculate(10000, 2);
        assertEquals(interest, 1000.0, 0);
    }

    @Test(expected = IllegalArgumentException.class)
    public void illegalCalculate() {
        interestCalculator.calculate(-10000, 2);
    }
}
```

Example: @BeforeClass, @AfterClass

```
public class SimpleInterestCalculatorJUnit4Tests {

    // One time setup for the whole tests
    @BeforeClass
    public static void myBeforeClassMethod() {
        System.out.println("myBeforeClassMethod - Set things up once for all");
    }

    // One time teardown for the whole tests
    @AfterClass
    public static void myAfterClassMethod() {
        System.out.println("myBeforeClassMethod - Clean things up once for all");
    }

}
```

Example: Timeout and Ignore

```
public class SimpleInterestCalculatorJUnit4Tests {

    @Test(timeout = 5000)
    public void testLengthyOperation() {
    }

    @Ignore("Ignore this test for now")
    @Test
    public void testTheWhatSoEverSpecialFunctionality() {
    }

}
```

Running JUnit4

- Most likely you are going to run within an IDE
- JUnit4 no longer supports TestRunner, instead you run at the command line
- Command line syntax

  java -cp <JUnit-jar> org.junit.runner.JUnitCore <Your-test-class>
- Example

  java -cp build\classes;C:\avase_junit\junitjarfile\junit-4.8.2.jar org.junit.runner.JUnitCore TestComputeClass

Demo:

TestComputeClass_annotation
1048_javase_junit.zip

TestSuite Support in JUnit 4.x

```
@RunWith(value=Suite.class)
@SuiteClasses(value={TestComputeClass.class, TestComputeClass2.class})
public class AllTests {

    // One time setup for the whole tests
    @BeforeClass
    public static void myBeforeClassMethod() {
        System.out.println("myBeforeClassMethod - Set things up once for all");
    }

    // One time teardown for the whole tests
    @AfterClass
    public static void myAfterClassMethod() {
        System.out.println("myAfterClassMethod - Clean things up once for all");
    }

}
```

Demo:

TestComputeClass_TestSuite_annotation 1048_javase_junit.zip

Best Practices

What should I test?

- Tests things which could break
- Tests should succeed quietly.
  - Don't print "Doing foo...done with foo!"
  - Negative tests, exceptions and errors
- What shouldn't I test
  - Don't test set/get methods
  - Don't test the compiler

Test Driven Development (TDD)

- Write your test first, or at least at the same time with your non-test code
- Create new tests to show bugs then fix the bug
- Test driven development says write the test then make it pass by coding to it.

Thank you!

Check JavaPassion.com Codecamps!
http://www.javapassion.com/codecamps
"Learn with Passion!"

Design Patterns for Testing

Designing for testing

- Separation of interface and implementation
  - Allows substitution of implementation to tests
- Factory pattern
  - Provides for abstraction of creation of implementations from the tests.
- Strategy pattern
  - Because FactoryFinder dynamically resolves desired factory, implementations are plugable

Design for testing - Factories

- `new` only used in Factory
- Allows writing tests which can be used across multiple implementations.
- Promotes frequent testing by writing tests which work against objects without requiring extensive setup
  - "extra-container" testing.

Design for testing - Mock Objects

- When your implementation requires a resource which is unavailable for testing
- External system or database is simulated.
- Another use of Factory, the mock implementation stubs out and returns the anticipated results from a request.

Testing with resources (EJB/DB)

- Use fixtures to request resource connection via factory, could be no-op.
- Use vm args or resource bundle to drive which factory is used.
- Data initialization/clearing handled by fixtures to preserve order independence of tests.