Fast Track to Java

Customized for Starbucks

Delivered by DevelopIntelligence

Unit Testing With JUnit

Objectives

At the end of this module you should be able to:

- Write and run tests using JUnit
- Write tests for code that should throw an exception
- write tests that will timeout if the test does not complete within a specified timescale
- Provide setup and cleanup code
- Work with doubles stubs and mocks
- Use Mockito to create mocks
- Verify calls and call order with Mockito

About Unit Testing

- Unit tests test program components in isolation
  - Errors in one unit do not corrupt test results for another
- Many simple, single-purpose tests
  - Make it easier to determine what’s really broken
  - Support changes/refactoring with confidence
  - Serve as documentation of the API of the units
- Must have tests for everything
  - Test Driven Development (TDD) insists that you do not write any code until you first have a broken test
  - There’s merit in this approach!
  - Tests improves ***both coding speed and code quality***

JUnit Overview

- Provides a framework for unit testing
  - Shouldn't be testing the testing framework in a live project
- Test are represented as special classes
  - Each test class typically contains multiple test methods
- `@Test` annotation labels a test method
- Tests are coded as "assertions"
  - JUnit predates Java's assertion mechanism
  - Provides a series of special assertion tests
- Tests (normally) pass if no exceptions arise

@Test Annotation

- Step 1 in creating a test is to create a test class
  - Parallel to your class under test
  - Usually in a separate, but parallel, package tree

```
import org.junit.Test;
[…]
@Test public void testSomething() { … }
```

Importing the Assertions

- Code the tests using JUnit assertions
  - JUnit provides myriad assertions as static methods in the `org.junit.Assert` class

```
import org.junit.Assert;
[…]
Assert.assertTrue([String], boolean);
```

  - Many examples/documentation use the static import

```
import static org.junit.Assert.*;
[…]
assertTrue([String], boolean);
```

Equality Assertion

- Most common test is for an expected value

`Assert.assertEquals([String], `*`expected`*`, `*`actual`*`)`

  - First argument is optional description text
  - Second argument is usually literal value
  - Third argument is computed value under test
- Test is performed using `.equals()` method
  - Ensure your objects under test provide proper equality comparison
- Overloads exist covering myriad types for expected/actual parameters

Equality of Floating Point

- Comparison of floating point types may be done with a degree of acceptable variation
  - Usually intended to ensure test does not fail because of rounding errors

```
Assert.assertEquals([String], double expect,
double actual, double variation)
Assert.assertEquals(Math.PI, 22.0/7.0, 0.01) //
passes
```

Example: Class Under Test

```
package undertest;
public class Account {
  private long balance;
  public static final int CHARGE = 35;

  public long getBalance() { return balance; }

  public long withdraw(long amount) {
    long newBalance = balance - amount;
    if (newBalance > 0) {
      balance = newBalance;
      return amount;
    } else {
      balance -= CHARGE;
      return 0;
    }
  }
}
```

Example: Test Class

```
package tests.undertest;
import org.junit.Assert;
import org.junit.Test;
import undertest.Account;

public class TestAccount {
  @Test public void testInitialBalance() {
    Assert.assertEquals("Initial balance should be zero",
      0, new Account().getBalance());
  }
  @Test public void testOverdrawn() {
    Account acc = new Account();
    Assert.assertEquals(0, acc.withdraw(10));
    Assert.assertEquals(-35, acc.getBalance());
  }
}
```

More Assertions

```
Assert.assertTrue([String], boolean)
Assert.assertFalse([String], boolean)

Assert.assertNull([String], Object)
Assert.assertNotNull([String], Object)
```

Array Equality Tests

- Testing the contents of two arrays are equivalent
  - Uses the `.equals()` test on each array element
  - `Assert.assertArrayEquals([String],` *`Many`*`[],` *`Many`*`[])`

- Also has a special form for testing arrays of floating point numbers with a rounding-error allowance

`Assert.assertArrayEquals([String], Floating[], Floating[],` ***`Floating`***`)`

More Assertions

- Test two references are the same
`Assert.assertSame([String], Object, Object)`
  - This uses == test instead of .equals()
- Give up and print a failure message
  - `Assert.fail([String])`
  - Use this to indicate not-yet-implemented test
  - Also to indicate “should not reach this point in test logic”
- Note, no need for “assertPass”—this is implied simply by not failing

Expected Exceptions

- Sometimes a test ***should*** throw an exception
  - This would normally be considered as a failure
  - “expected“ states that the exception should arise
  - If exception is ***not*** thrown, test fails
  - Must use throws construction on test method
  - Tests should test exactly one condition, so expected only allows one exception

```
@Test(expected=SomeException.class)
public void testOutOfRange
  throws SomeException { … }
```

Timeout

- Some tests might deadlock or otherwise fail to return in a timely fashion
- Use the timeout option to force a test to fail if it takes too long
  - Specify timeout in milliseconds
- ***Do not use this for performance testing***

```
@Test(timeout=300)
public void testMethod() { … }
```

@Before and @After

- Independent tests imply a new object under test for each test
  - Tests might all need the same setup and teardown

```
public class MyTests {
  @Before public void setup() { … }
  @After public void clean () { … }
  @Test public void testA() { … }
  @Test public void testB() { … }
}
```

@BeforeClass and @AfterClass

- Independent tests actually involve creating a new instance of the Test class for every test
  - Allows use of instance variables should you wish
  - Makes sharing data and configuration (not code) between tests deliberately difficult
  - Can use statics if necessary (try to avoid it!)
- Class initialization and clean up is supported using `@BeforeClass` and `@AfterClass` annotations

@BeforeClass and @AfterClass

```
public class MyTests {
   @BeforeClass
   public void setupClass() { … }
   @AfterClass
   public void cleanClass() { … }
   @Test public void testA() { … }
   @Test public void testB() { … }
}
```

Running Tests

- Single tests in Eclipse
  - Right click class, run-as JUnit test
- Group tests in Eclipse
  - Right click package, run-as JUnit test
- Test runners
  - JUnit package provides Parameterized, and others
- Test from a command line

```
java org.junit.runner.JUnitCore
TestClass TestClass1 …
```

Fakes, Doubles, Stubs & Mocks

- To maintain the independence of our tests, we must avoid dependence on supporting objects
  - This is something of a paradox, for example, how can we test something that processes results from a database, without using the database?
- Usual approach is to replace the supporting object with something else
  - Must implement the same interface
  - Unit under test must not hard-code the supporting object
  - Unit under test must be coded to use supporting object by interface
- Replacement objects have various names fakes, doubles, stubs, or mocks

Replacing Supporting Objects

- Doubles can simply be coded to interfaces
  - Unit testing seeks to avoid untested test harnesses
- Mocking tools have been created to allow a declarative approach to creating replacements
  - These tools seek to minimize chances of error
  - Syntax is intended to make the function of the mock “obvious”
- Mockito, JMockit, EasyMock

Summary

In this module, we covered:

- How to write and run tests using JUnit
- How to write tests for code that should throw an exception
- How to write tests that will timeout if the test does not complete within a specified timescale
- How to provide setup and cleanup code
- About doubles stubs and mocks