JUnit 4 – Java Testing Framework

Kurt Schmidt

Dept. of Computer Science, Drexel University

December 28, 2016

Examples from these slides can be found in
tux: kschmidt/public_html/CS265/Labs/Java/Junit

Intro

JUnit

- These slides assume you've read a bit about testing
- In JUnit 4, we don't need any special classes
- We simply create test classes
  - Use annotations to identify the roles of the various methods
  - Add methods to test behaviors
- Compatibility with JUnit 3 is, apparently, maintained

Annotations

Annotations

Since JUnit 4

`@Test`	Method is a test method
`@Test(timeout=`*n*`)`	Will fail if it takes longer than *n* ms
`@Test(expected=`*e*`)`	Fail unless exception *e* is thrown
`@BeforeClass`	Method will be invoked ! once, before tests
`@Before`	Method called before each test
`@After`	Method called after each test
`@AfterClass`	Called ! once, after tests
`@Ignore`	Indicates test should be ignored

Assertions

`Assert` Static Methods

- Found in `org.junit.Assert`[1]
- All are overloaded to take an optional message (`String`) as a first argument
  - `assertTrue( boolean )`
  - `assertFalse( boolean )`
  - `assertEquals( T, T )`
    - Overloaded to take any primitives, or `Object`
    - If `Object`, uses `equals()`

[1] Must use Java 5's `import static` feature. I have no idea why.

`Assert` Methods – cont.

- These expect references to `Ojbect`
  - `assertNull( Object )`
  - `assertNotNull( Object )`
  - `assertSame( Object, Object )`
    - Checks to see that both refer to same object
  - `assertNotSame( Object, Object )`
  - `fail()`
    - Dumps the testing, with optional message

Example

Example – `Money` Class

- Simple class, stores amount of money, a scalar, and a currency
- A couple simple behaviors:
  - `equals` – Overridden simply for the sake of the example
  - `add` – For adding two amounts
- Plus the usual getters
- Note, instance of `Money` is immutable

Money Class


```
public class Money {
   private int fAmount;
   private String fCurrency;

   ...

   public boolean equals( Object anObject ) {
      if( anObject instanceof Money ) {
         Money aMoney = (Money)anObject;
         return aMoney.currency().equals( currency() )
            && amount() == aMoney.amount();
      }
      return false;
   }

   public Money add( Money rhs ) {
      if( ! rhs.currency().equals( fCurrency ))
         return null;
      return new Money( amount()+rhs.amount(), currency() );
   }
} // class Money
```

Writing a Test Case


```
import java.io.* ;

import static org.junit.Assert.* ;
import org.junit.* ;

public class MoneyTest extends TestCase
{
   private Money m12CHF;
   private Money m14CHF;
   private Money md13CHF;
   ...
}
```

Setting Up

- `@BeforeClass` tags any static methods that are to be run before this suite is run
- `@Before` tags any methods that are to be run before each test

```
public class MoneyTest extends TestCase {
   ...

   @Before
   public void setUp() {
      m12CHF= new Money( 12, "CHF" );
      m14CHF= new Money( 14, "CHF" );
      md13CHF = new Money( -13, "CHF" );
   }

   @BeforeClass
   public static void silly()
   { System.err.println( "### Starting run...\n" ) ; }

   ...
}
```

Tear Down, Clean up

- `@AfterClass` tags any static methods that are to be run after this suite is run
- `@After` tags any methods that are to be run after each test

```
public class MoneyTest {
   ...

   @After // After every test
   public void tearDown()
   {  System.err.println( "### Done test..." ) ; }

   @AfterClass
   public static void yllis()
   {  System.err.println( "\n### Done run..." ) ; }
```

Adding Tests

- Write black-box tests for a given behavior first
- As you implement behavior to the class being tested, add more tests to your test case.
  - Annotate with `@Test`
  - Public, non-static method
  - Takes no arguments
  - Return type of `void`
  - Use various Assert methods to access various hooks into the framework

Example Tests

- Note, these examples should not be considered to be adequate tests

```
public class MoneyTest {
   ...
   @Test
   public void testEquals() {
      Money expected = new Money( 12, "CHF" );
      assertEquals( expected, m12CHF );
      assertNotSame( expected, m12CHF );
      assertFalse( m12CHF.equals( m14CHF ));
      ...
   }
   ...
}
```

Test Timeout

- `@Test` annotation optionally takes a `timeout` argument, in ms.

```
public class MoneyTest {
   ...
   @Test(timeout=1000)
   public void testAdd() {
      Money expected26 = new Money( 26, "CHF" );
      Money expectedd1 = new Money( -1, "CHF" );

      assertNotNull( expected26 );

      Money result26 = m12CHF.add( m14CHF );
      assertNotNull( result26 );
      assertEquals( expected26, result26 );
      ...
   }
   ...
}
```

Test Exceptions

- Generally, JUnit 4 will fail, if it encounters an uncaught exception
- We can test that a particular exception is thrown

```
public class MoneyTest {
   ...
   @Test(expected=Money.IncompatibleUnitException.class)
   public void crashAdd()
   {
         // Attempt to add Suisse francs to zloty
      Money m = new Money( 120, "PLN" ) ;

      m12CHF.add( m ) ;
   }
   ...
}
```

Ignoring Tests

- `@Ignore` will cause a test to be ignored
  - It'll still be reported

```
public class MoneyTest {
   ...
   @Ignore( "Feature not implemented yet" )
   @Test
   public void testConvert()
   {
      Money old = new Money( 200, "USD" ) ;
      Money pocket = old.convert( "PLN" ) ;
   }
   ...
}
```

Running Your Tests From the Command Line

- You can invoke the test runner, pass class(es) containing tests as arguments

```
$ export CLASSPATH="/usr/share/java/junit4.jar:."
$ javac Money.java MoneyTest.java
$ java org.junit.runner.JUnitCore MoneyTest
```

```
JUnit version 4.12
.I..
Time: 0.007

OK (3 tests)
```

Running Your Tests from Within Java


```
public class MoneyTest {
   ...
   public static void main( String [] args )
   {
      org.junit.runner.JUnitCore.runClasses( MoneyTest.class ) ;
   }
   ...
}
```

- Then, just run it:

```
$ javac Money.java MoneyTest.java
$ java MoneyTest
```

Running Your Tests from Ant

build.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project default='test' basedir='.'>
   <path id="project.class.path">
      <pathelement path='${CLASSPATH}'/>
      <pathelement location='/usr/share/java/junit4.jar'/>
      <pathelement location='.'/>
   </path>
   ...
   <target name='test' depends='compile,MoneyTest'>
      <junit>
         <classpath refid="project.class.path"/>
         <formatter type='plain'/>
         <test name='MoneyTest'/>
      </junit>
   </target>
</project>
```

Running Your Tests from Ant

- Report is in a text file

```
$ ant test
$ cat TEST-MoneyTest.txt
```

```
Testsuite: MoneyTest
Tests run: 4, Failures: 0, Errors: 0, Skipped: 1, Time elapsed: 0.059 se

Testcase: testAdd took 0.002 sec
Testcase: testConvert took 0 sec
   SKIPPED: Feature not implemented yet
Testcase: testEquals took 0 sec
Testcase: crashAdd took 0 sec
```

Gathering Test Cases Together

- You can organise test classes into one class:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({
   CompressionTest.class,
   SparkTest.class,
   SomeOtherTest.class })
public class SuiteTests { }
```

- Run this as you ran your test classes, above

Hints

Hints from Prof. Noll[1]

Tests should be silent. Do not under any circumstances use `System.out.println` *in any test method. Rather, use assertions.*

Before you add a method to your production class, think about the pre-conditions and post-conditions for the method... Then, capture the pre-/post-conditions as initialization code and assertions in a unit test method: initialize the pre-conditions, call the method, assert the post-conditions. This ... ensures that you understand what the method is supposed to do before you write it.

[1] Santa Clara University

Hints from Prof. Noll[1] (cont.)

When you are tempted to put `System.out.println` in your production code, instead write a test method. This will help to clarify your design, and increase the coverage of your unit tests. It also prevents scroll blindness, as the tests say nothing until a failure is detected.

Don't put `System.out.println` in your production code. If you want to do this to observe the behavior of your program, write a unit test to assert its behavior instead. If you need to print to stdout as part of the program's functionality, pass a PrintWriter or output stream to those methods that do printing. Then, you can easily create unit tests for those methods.

[1] Santa Clara University