JUnit

A framework which provides hooks for easy testing of your Java code, as it's built

Note: The examples from these slides can be found in ~kschmidt/public_html/CS265/Labs/Java/Junit

JUnit

- This assumes that you've read a bit about testing; I'm not going to lecture here
- The heart of this is a test case, a class which contains tests
  - A test is a method that test methods that you've written
  - The framework is found in junit.jar
  - On the CS machines, currently, is located at /usr/share/java/junit.jar

The `TestCase` class

- Each test case is a class you provide that extends (inherits from) `TestCase`
- Contains:
  - member data
  - special admin-type methods:
    - `setUp()` – run before each test
    - `tearDown()` – run after each test
    - `suite()` – (static), a method that, basically, lists the tests to be added to the framework
  - the tests
    - methods that take no arguments
    - often start with or contain the string "Test" (for older versions of the framework)

Example – Money class

- The files for these examples can be found in:

  `~kschmidt/public_html/CS265/Labs/Java/Junit`

- `MoneyTest` written to test class `Money`

Example – extending TestCase, c'tor

```
import junit.framework.*;

public class MoneyTest extends TestCase
{
   public MoneyTest( String name ) {
    super( name );
   }

   ...
}
```

Example - setup

- Add some Money objects to use in our tests:

```
public class MoneyTest extends TestCase
{
   public MoneyTest( String name ) {
    super( name );
   }

   private Money m12CHF;
   private Money m12CHF;

    protected void setUp() {
      m12CHF= new Money( 12, "CHF" );
      m14CHF= new Money( 14, "CHF" );
    }

   ...
}
```

Adding Tests

- When you implement a method, the test for that method should actually be written first
- Add a corresponding "test" method to your `TestCase` class
  - Takes no arguments
  - returns `void`
  - use various `Assert` methods to access hooks into the framework (pass/fail, logging)

JUnit: `Testcase` & `Assert`

- To use classes from the JUnit framework (TestCase, Assert, etc):

```
import junit.framework.*
```

`Assert` static methods

- Static methods of the class `Assert`
- All are overloaded to take an optional message (a `String`) as the first argument
  - `assertTrue( boolean condition )`
  - `assertFalse( boolean condition )`
  - `assertEquals( expected, actual )`
    - overloaded to take any primitives, or anything derived from `Object`
    - Note, if subtypes of `Object`, need to override `equals()`

Assert methods (cont)

- assertNull(Object)
- assertNotNull(Object)
- assertSame(Object expected, Object actual)
  - Checks to see that expected and actual refer to the same object (so, of course, are equal)
- assertNotSame(Object expected, Object actual)
- fail()
  - Just dumps the testing, w/the optional msg

Example – override `equals()`

- We need to provide `Money.equals()`:

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

Test equals

- Add to MoneyTest:

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

Test `Money.add()`

- Add to MoneyTest:

```
public void testAdd() {
   Money expected= new Money( 26, "CHF" );
   Money result= m12CHF.add( m14CHF );
   Assert.assertEquals( expected, result );

   expected = new Money( 1, "CHF" );
   result = m12CHF.add( md13CHF );
   Assert.assertEquals( expected, result );

   result = md13CHF.add( m12CHF );
   Assert.assertEquals( expected, result );
}
```

Identifying tests – `suite()`

- You've written some tests
- They need to be identified to the testing framework
- Override the `suite()` static method
  - Create a `TestSuite` object
  - Use its `addTest()` method to add tests you want to run

Example – `suite()`

```
   // adding each test
   // you can just add the tests you want
public static Test suite() {
   TestSuite suite= new TestSuite();
   suite.addTest( new MoneyTest( "testEquals" ));
   suite.addTest( new MoneyTest( "testAdd" ));
   return suite;
}
```

Adding all tests

- You can write a `suite()` method that uses reflection to add *all* of the `test`*`XXX`*`()` methods in the `TestCase` class to the Suite:

```
public static Test suite() {
   return new TestSuite( MoneyTest.class );
}
```

Testing your test case

- As a sanity check, after writing a test or two, you might want to make a little `main` method for the `TestCase` class:

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

  - This will set up and run the tests
  - Make sure you didn't make any errors in the testing framework
  - (Remember to add `junit.jar` to `$CLASSPATH` to run this from the command line)

Hints (from Prof. Noll at Santa Clara Univ.)

- 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: what needs to be in place before the method can be called, and what is supposed to have happened after the method returns? 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 accomplishes two things: first, it ensures that you understand what the method is supposed to do *before* you write it. Second, it provides a test for the method that is available as soon as the method is ready to test.

Hints (cont.)
(from Prof. Noll at Santa Clara Univ.)

- 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 (see also above). 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.

Example – build.xml

```
<target name='test' depends='compile'>
  <junit>
   <formatter type='plain'/>
   <test name='MoneyTest'/>
  </junit>
</target>
```

Ant hook for Junit

- You can have Ant run these tests, as you write and compile each method
- Add a target to build.xml
  - depends on the class and your `TestCase` being compiled
  - Might modify the classpath to include junit.jar
  - Have a jar action that describes the tests you want to run