

JUnit

- A unit test framework for Java
 - Authors: Erich Gamma, Kent Beck
- Objective:
 - “If tests are simple to create and execute, then programmers will be more inclined to create and execute tests.”

Introduction

- What do we need to do automated testing?
 - Test script
 - Actions to send to system under test (SUT).
 - Responses expected from SUT.
 - How to determine whether a test was successful or not?
 - Test execution system
 - Mechanism to read test scripts, and connect test case to SUT.
 - Keeps track of test results.

Test case verdicts

- A **verdict** is the declared result of executing a single test.
- **Pass**: the test case achieved its intended purpose, and the software under test performed as expected.
- **Fail**: the test case achieved its intended purpose, but the software under test did not perform as expected.
- **Error**: the test case did not achieve its intended purpose.
 - Potential reasons:
 - An unexpected event occurred during the test case.
 - The test case could not be set up properly

A note on JUnit versions...

- The current version is 4.3.1, available from Mar. 2007
 - To use JUnit 4.x, you **must** use Java version 5 or 6
- JUnit 4, introduced April 2006, is a significant (i.e. not compatible) change from prior versions.
- **JUnit 4 is used in this presentation.**
- Much of the JUnit documentation and examples currently available are for JUnit 3, which is slightly different.
 - JUnit 3 can be used with earlier versions of Java (such as 1.4.2).
 - The junit.org web site shows JUnit version 4 unless you ask for the old version.
 - Eclipse (3.2) gives the option of using JUnit 3.8 or JUnit 4.1, which are both packaged within Eclipse.

What is a JUnit Test?

- A test “script” is just a collection of Java methods.
 - General idea is to create a few Java objects, do something interesting with them, and then determine if the objects have the correct properties.
- What is added? Assertions.
 - A package of methods that checks for various properties:
 - “equality” of objects
 - identical object references
 - null / non-null object references
 - The assertions are used to determine the test case verdict.

When is JUnit appropriate?

- As the name implies...
 - for unit testing of small amounts of code
- On its own, it is not intended for complex testing, system testing, etc.
- In the test-driven development methodology, a JUnit test should be written first (before any code), and executed.
 - Then, implementation code should be written that would be the minimum code required to get the test to pass – and no extra functionality.
 - Once the code is written, re-execute the test and it should pass.
 - Every time new code is added, re-execute all tests again to be sure nothing gets broken.

A JUnit 4 Test Case

```
/** Test of setName() method, of class Value */
```

```
@Test
```

```
public void createAndSetName()
```

```
{
```

```
    Value v1 = new Value();
```

```
    v1.setName( "Y" );
```

```
    String expected = "Y";
```

```
    String actual = v1.getName();
```

```
    Assert.assertEquals( expected, actual );
```

```
}
```

A JUnit 4 Test Case

```
/** Test of setName() method, of class Value */
```

```
@Test
```

```
public void createA  
{
```

```
    Value v1 = new Value();
```

```
    v1.setName( "Y" );
```

```
    String expected = "Y";
```

```
    String actual = v1.getName();
```

```
    Assert.assertEquals( expected, actual );
```

```
}
```




Identifies this Java method
as a test case, for the test runner

A JUnit 4 Test Case

```
/** Test of setName() method, of class Value */
```

```
@Test  
public void createAndSetName()  
{  
    Value v1 = new Value();  
  
    v1.setName( "Y" );  
  
    String expected = "Y";  
    String actual = v1.getName();  
  
    Assert.assertEquals( expected, actual );  
}
```

Objective:
confirm that **setName**
saves the specified name in
the **Value** object



A JUnit 4 Test Case

```
/** Test of setName() method, of class Value */
```

```
@Test
```

```
public void createAndSetName()
```

```
{
```

```
    Value v1 = new Value();
```

```
    v1.setName( "Y" );
```

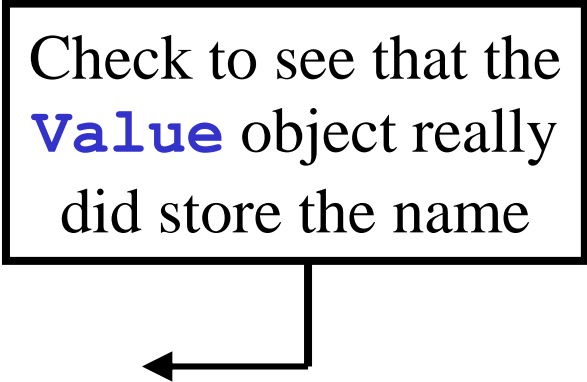
```
    String expected = "Y"
```

```
    String actual = v1.getName();
```

```
    Assert.assertEquals( expected, actual );
```

```
}
```

Check to see that the **Value** object really did store the name

A rectangular box with a black border containing the text "Check to see that the Value object really did store the name". A vertical line extends downwards from the bottom center of the box, ending in a horizontal arrow pointing to the left.


A JUnit 4 Test Case

```
/** Test of setName() method, of class Value */
```

```
@Test  
public void createAndSetName()  
{  
    Value v1 = new Value();  
  
    v1.setName( "Y" );  
  
    String expected = "Y";  
    String actual = v1.getName();  
  
    Assert.assertEquals( expected, actual );  
}
```

We want **expected** and **actual** to be equal.

If they aren't, then the test case should fail.



Assertions

- Assertions are defined in the JUnit class **Assert**
 - If an assertion is true, the method continues executing.
 - If any assertion is false, the method stops executing at that point, and the result for the test case will be **fail**.
 - If any other exception is thrown during the method, the result for the test case will be **error**.
 - If no assertions were violated for the entire method, the test case will **pass**.
- All assertion methods are **static** methods

Assertion methods (1)

- Boolean conditions are true or false

assertTrue(condition)

assertFalse(condition)

- Objects are null or non-null

assertNull(object)

assertNotNull(object)

- Objects are identical (i.e. two references to the same object), or not identical.

assertSame(expected, actual)

– true if: **expected == actual**

assertNotSame(expected, actual)

Assertion methods (2)

- “Equality” of objects:

assertEquals(expected, actual)

– valid if: **expected.equals(actual)**

- “Equality” of arrays:

assertArrayEquals(expected, actual)

– arrays must have same length

– for each valid value for **i**, check as appropriate:

assertEquals(expected[i],actual[i])

or

assertArrayEquals(expected[i],actual[i])

- There is also an unconditional failure assertion **fail()** that **always** results in a fail verdict.

Assertion method parameters

- In any assertion method with two parameters, the first parameter is the **expected** value, and the second parameter should be the **actual** value.
 - This does not affect the comparison, but this ordering is assumed for creating the failure message to the user.
- Any assertion method can have an additional **String** parameter as the first parameter. The string will be included in the failure message if the assertion fails.
 - Examples:
 - fail(message)**
 - assertEquals(message, expected, actual)**

Equality assertions

- **assertEquals(a,b)** relies on the **equals()** method of the class under test.
 - The effect is to evaluate **a.equals(b)**.
 - It is up to the class under test to determine a suitable equality relation. JUnit uses whatever is available.
 - Any class under test that does **not** override the **equals()** method from class **Object** will get the default **equals()** behaviour – that is, object identity.
- If **a** and **b** are of a primitive type such as **int**, **boolean**, etc., then the following is done for **assertEquals(a,b)** :
 - **a** and **b** are converted to their equivalent object type (**Integer**, **Boolean**, etc.), and then **a.equals(b)** is evaluated.

Floating point assertions

- When comparing floating point types (**double** or **float**), there is an additional required parameter **delta**.
- The assertion evaluates

`Math.abs(expected – actual) <= delta`

to avoid problems with round-off errors with floating point comparisons.

- Example:

`assertEquals(aDouble, anotherDouble, 0.0001)`

Organization of JUnit tests

- Each method represents a single test case that can independently have a verdict (pass, error, fail).
- Normally, all the tests for one Java class are grouped together into a separate class.
 - Naming convention:
 - Class to be tested: **Value**
 - Class containing tests: **ValueTest**

Running JUnit Tests (1)

- The JUnit framework does not provide a graphical test runner. Instead, it provides an API that can be used by IDEs to run test cases and a textual runner that can be used from a command line.
- Eclipse and Netbeans each provide a graphical test runner that is integrated into their respective environments.

Running JUnit tests (2)

- With the runner provided by JUnit:
 - When a class is selected for execution, all the test case methods in the class will be run.
 - The order in which the methods in the class are called (i.e. the order of test case execution) is **not predictable**.
- Test runners provided by IDEs **may** allow the user to select particular methods, or to set the order of execution.
- It is good practice to write tests which are independent of execution order, and that are without dependencies on the state of any previous test(s).

Test fixtures

- A test fixture is the context in which a test case runs.
- Typically, test fixtures include:
 - Objects or resources that are available for use by any test case.
 - Activities required to make these objects available and/or resource allocation and de-allocation: “setup” and “teardown”.

Setup and Teardown

- For a collection of tests for a particular class, there are often some repeated tasks that must be done prior to each test case.
 - Examples: create some “interesting” objects to work with, open a network connection, etc.
- Likewise, at the end of each test case, there may be repeated tasks to clean up after test execution.
 - Ensures resources are released, test system is in known state for next test case, etc.
 - Since a test case failure ends execution of a test method at that point, code to clean up **cannot** be at the end of the method.

Setup and Teardown

- Setup:
 - Use the **@Before** annotation on a method containing code to run before each test case.
- Teardown (*regardless of the verdict*):
 - Use the **@After** annotation on a method containing code to run after each test case.
 - These methods will run even if exceptions are thrown in the test case or an assertion fails.
- It is allowed to have any number of these annotations.
 - All methods annotated with **@Before** will be run before each test case, but they may be run in *any* order.

Example: Using a file as a text fixture

```
public class OutputTest
{
    private File output;

    @Before public void createOutputFile()
    {
        output = new File(...);
    }

    @After public void deleteOutputFile()
    {
        output.delete();
    }

    @Test public void test1WithFile()
    {
        // code for test case objective
    }

    @Test public void test2WithFile()
    {
        // code for test case objective
    }
}
```


Method execution order

1. **createOutputFile()**

2. **test1WithFile()**

3. **deleteOutputFile()**

4. **createOutputFile()**

5. **test2WithFile()**

6. **deleteOutputFile()**

- Assumption: **test1WithFile** runs before **test2WithFile**—which is not guaranteed.

Once-only setup

- It is also possible to run a method **once only** for the entire test class, **before** any of the tests are executed, and prior to any **@Before** method(s).
- Useful for starting servers, opening communications, etc. that are time-consuming to close and re-open for each test.
- Indicate with **@BeforeClass** annotation (can only be used on **one** method, which must be **static**):

```
@BeforeClass public static void anyNameHere()  
  
{  
  
    // class setup code here  
  
}
```

Once-only tear down

- A corresponding once-only cleanup method is also available. It is run after all test case methods in the class have been executed, and after any **@After** methods
- Useful for stopping servers, closing communication links, etc.
- Indicate with **@AfterClass** annotation (can only be used on **one** method, which must be **static**):

```
@AfterClass public static void anyNameHere()  
  
{  
  
  // class cleanup code here  
  
}
```

Exception testing (1)

- Add parameter to **@Test** annotation, indicating that a particular class of exception is expected to occur during the test.

```
@Test(expected=ExpectedTypeOfException.class)  
public void testException()  
{  
    exceptionCausingMethod();  
}
```

- If no exception is thrown, or an unexpected exception occurs, the test will fail.
 - That is, reaching the end of the method with no exception will cause a test case failure.
- Testing contents of the exception message, or limiting the scope of where the exception is expected requires using the approach on the next slide.

Exception testing (2)

- Catch exception, and use **fail()** if not thrown

```
public void testException()
{
    try
    {
        exceptionCausingMethod();

        // If this point is reached, the expected
        // exception was not thrown.

        fail("Exception should have occurred");
    }
    catch ( ExpectedTypeOfException exc )
    {
        String expected = "A suitable error message";
        String actual = exc.getMessage();
        Assert.assertEquals( expected, actual );
    }
}
```

JUnit 3

- At this point, migration is still underway from JUnit 3 to JUnit 4
 - Eclipse 3.2 has both
 - The Eclipse test and performance tools platform does not yet work with JUnit 4.
 - Netbeans 5.5 has only JUnit 3.
- Within the JUnit archive, the following packages are used so that the two versions can co-exist.
 - JUnit 3: **junit.framework.***
 - JUnit 4: **org.junit.***

Topics for another day...

- Differences between JUnit 3 and JUnit 4
- More on test runners
- Parameterized tests
- Tests with timeouts
- Test suites