JUnit - A Whole Lot of Testing Going On

Advanced Topics in Java

Khalid Azim Mughal
khalid@ii.uib.no
http://www.ii.uib.no/~khalid

Version date: 2006-09-04

Overview

- Software Testing
- JUnit Overview
- Installing JUnit
- Writing Unit Tests
- Running Unit Tests
- Test Failures and Test Errors
- Running Unit Tests: GUI mode
- The `assert`*Fact*`()` Methods
- Invoking Unit Test Methods
- Tests Using `setUp()` and `tearDown()`
- Test Suites
- Running Selected Unit Tests
- Multiple Test Suites
- Repeating Tests
- Exception Handling
- Running Tests Concurrently
- Organizing Tests in Packages
- Running the Same Unit Test Method Repeatedly
- Encapsulating Common Behavior of Tests

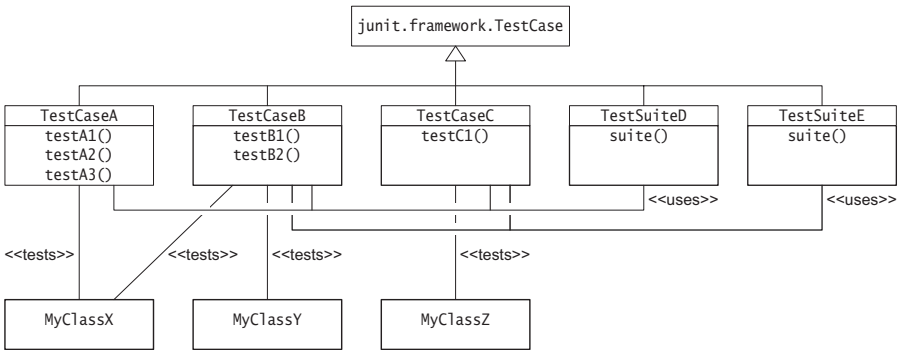
Software Testing

- *Testing* is the practice of ensuring that the software functions according to customer requirements.
  - *Unit testing* refers to testing a unit of code (usually methods in the interface of a class) extensively in all possible ways.
  - *Regression testing* refers to retesting the software exhaustively to make sure that any new changes introduced have not broken the system.
- *Refactoring* is the practice of restructuring existing code for simplicity and clarity, without changing its external behavior (`http://www.refactoring.com/`).
- Refactoring and testing are two vital activities emphasized in *Extreme Programming* (`http://www.extremeprogramming.org/`).
- Refactoring and testing are necessary throughout the software development process.
- *Automated testing* is preferable as it allows the software to be tested as and when changes are made.
- *JUnit* is a regression testing framework for implementing and running unit tests in Java.

JUnit Overview

- Framework for automating unit testing.
  - Allows *creation* of unit tests.
  - Allows *execution* of unit tests.
- A JUnit *test case* is a Java class containing one or more *unit tests*.
  - A *test case* is a subclass of the `junit.framework.TestCase` class (also called a *fixture*).
  - A *unit test* is typically a `public`, `void`, no-parameter method named `test`*Something*.
  - A *test suite* is a collection of *test cases*.
- Tests can be run as individual test cases or entire test suites.
  - Tests run in *batch mode*, i.e. there is no interaction with the user during the running of the tests.
  - Outcome of running a unit test is either pass or fail.

Implementing Tests



- The class diagram shows 3 test cases (`TestCaseA`, `TestCaseB`, `TestCaseC`) and 2 test suites (`TestSuiteD`, `TestSuiteE`).
- `TestCaseA` tests `MyClassX` and implements 3 unit tests: `testA1()`, `testA2()` and `testA3()`.
- `TestCaseB` tests `MyClassX` and `MyClassY`, and implements 2 unit tests: `testB1()` and `testB2()`.
- `TestCaseC` tests `MyClassZ` and implements 1 unit test: `testC1()`.
- `TestSuiteD` runs the tests in all the 3 test cases.
- `TestSuiteE` runs the tests in the test cases `TestCaseB` and `TestCaseC`.

Installing JUnit on Windows

- Download Junit (`Junit3.8.1.zip`) from the website `http://junit.org/`.
- Unzip the zip file there you want to install JUnit.
  - This action creates a directory with the JUnit installation, say this directory is called `Junit3.8.1`.
- Change the name of this directory to `junit_home`.
  - This way a newer version of JUnit can be installed without having to go through the rest of the procedure.
  - Say the full path of this directory is `C:\junit_home`.
- Add the path of the `junit.jar` file in the `junit_home` directory to your `CLASSPATH` environment variable, i.e. the path `C:\junit_home\junit.jar` should be added to the `CLASSPATH` variable.

Writing Simple Unit Tests with JUnit

Procedure for writing a test case:

1. Declare a subclass of the `junit.framework.TestCase` class.
   - The name of the class can be any legal name which is meaningful in the given context.
2. Declare one or more unit tests by implementing `test`*Something*`()` methods in the subclass.
   - The name of the unit test method must have the prefix `test`, and *Something* usually signifies what aspect the method will test.
   - The JUnit framework requires that a test case has at least one unit test method.
3. Call `assert`*Fact*`()` methods in each `test`*Something*`()` method to perform the actual testing.
   - Various overloaded `assert`*Fact*`()` methods are inherited by the subclass to test different conditions.
   - Typically, the `assertEquals()` method is used to assert that its two arguments are equal.

Example I

- An object of the class `Comparison` stores a secret number.
- The method `compare()` determines whether its argument is equal, greater or less than the secret number.
- We will create tests to determine whether the `compare()` method is implemented correctly.

```
/** Comparison.java */
public class Comparison {
    /** The number to compare with */
    private final int SECRET;
    /** @param number the number to compare with */
    public Comparison(int number) {
         SECRET = number;
    }
    /**
     * Compares the guess with the number stored in the object.
     * @return  the value 0 if the guess is equal to the secret;
                the value 1 if the guess is greater than the secret;
                and the value -1 if the guess is less than the secret.
     */
    public int compare(int guess) {
        int status = 0;
        if (guess > SECRET)
            status = 1;
        else if (guess < SECRET)
            status = -1;
        return status;
    }
}
```


Writing Unit Tests

- The class `TestComparison` defines a test case consisting of 4 unit tests implemented by the following methods:

```
public void testEqual()   { ... }
public void testGreater() { ... }
public void testLess()    { ... }
public void testAll()     { ... }
```

  - Each unit test method tests the result returned by the `compare()` method for different guesses.
  - The `testAll()` method tests all the conditions.
- In each unit test method, we create an object of class `Comparison` and test different conditions using the `assertEquals()` method.
  - The first argument of the `assertEquals()` method is the *expected result*.
  - The *actual result* returned by the evaluation of the second argument is compared for equality with the expected result by the `assertEquals()` method to determine whether this assertion condition holds.
- Make sure that the `junit.jar` file with the JUnit framework is in the classpath of the compiler when compiling test case classes.

```
/** TestComparison.java */
import junit.framework.TestCase;
public class TestComparison extends TestCase {           // (0) Test case
    public TestComparison(String testMethodName) { super(testMethodName); }
    public void testEqual() {                             // (1) Unit test
        Comparison firstOperand = new Comparison(2004);
        assertEquals( 0, firstOperand.compare(2004));
    }
    public void testGreater() {                           // (2) Unit test
        Comparison firstOperand = new Comparison(2004);
        assertEquals( 1, firstOperand.compare(2010));
    }
    public void testLess() {                              // (3) Unit test
        Comparison firstOperand = new Comparison(2004);
        assertEquals(-1, firstOperand.compare(2000));
    }
    public void testAll() {                               // (4) Unit test
        Comparison firstOperand = new Comparison(2004);
        assertEquals( 0, firstOperand.compare(2004));
        assertEquals( 1, firstOperand.compare(2010));
        assertEquals(-1, firstOperand.compare(2000));
    }
}
```


Running Unit Tests: Text mode

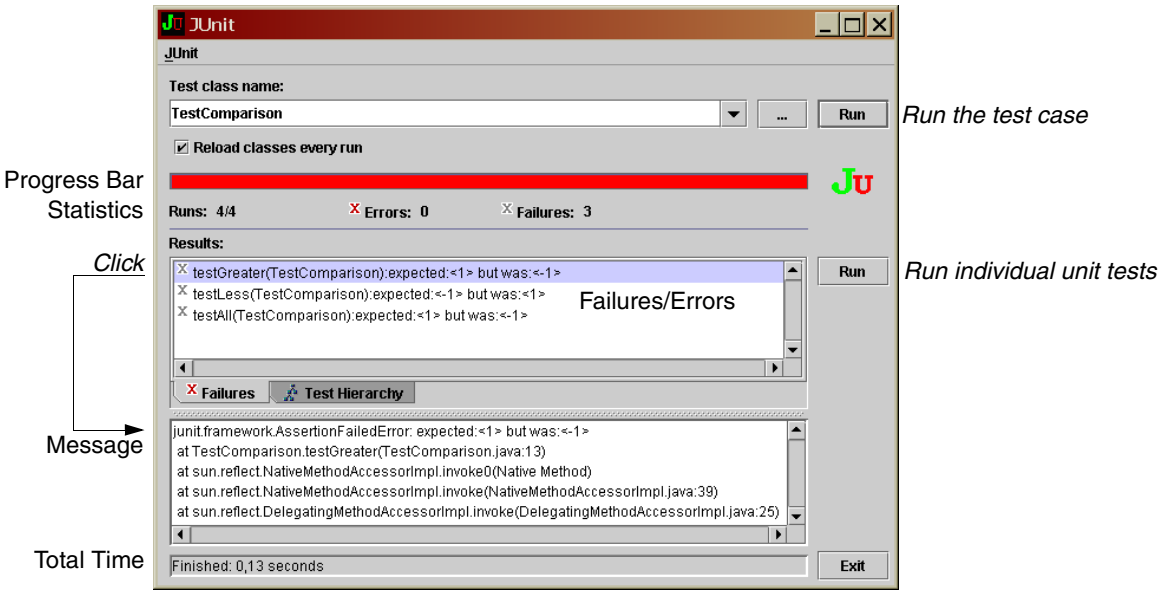
```
>java junit.textui.TestRunner TestComparison
..F.F.F                    <========== Progress line
Time: 0,01                 <========== Total time
There were 3 failures:     <========== Failures/errors
1) testGreater(TestComparison)junit.framework.AssertionFailedError: expected:<1> but was:<-1>
        at TestComparison.testGreater(TestComparison.java:13)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
2) testLess(TestComparison)junit.framework.AssertionFailedError: expected:<-1> but was:<1>
        at TestComparison.testLess(TestComparison.java:18)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
3) testAll(TestComparison)junit.framework.AssertionFailedError: expected:<1> but was:<-1>
        at TestComparison.testAll(TestComparison.java:24)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)

FAILURES!!!
Tests run: 4,  Failures: 3,  Errors: 0    <========== Test Statistics
```

Test Failures and Test Errors

- A *test failure* occurs when an `assert`*Fact*`()` method fails.
  - Indicated by a `junit.framework.AssertionFailedError` in the output, indicating the expected and the actual result.
- A *test error* occurs when a `test`*Something*`()` method throws an exception.
  - Indicated by the appropriate exception in the output.
- Execution flow in case of failure or error:
  - The current unit test method is aborted.
  - Execution continuing with the next unit test method, if any.

Running Unit Tests: GUI mode

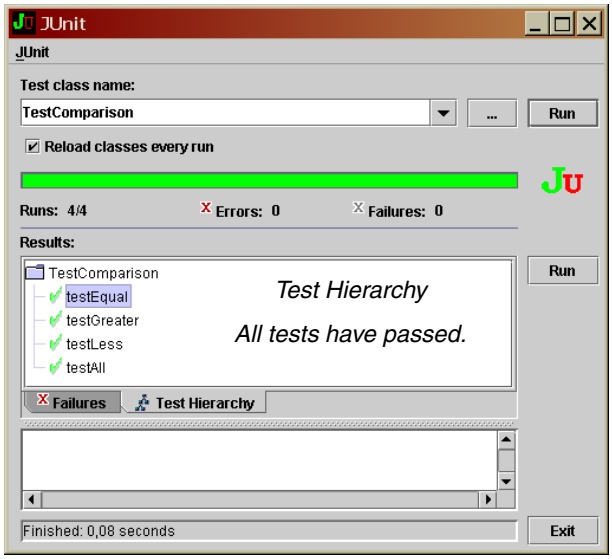
```
>java junit.swingui.TestRunner TestComparison
```



Running Unit Tests: GUI mode (cont.)

- Running the tests after correcting the program:

```
...
public int compare(int guess) {
    int status = 0;
    if (guess > SECRET)
        status = 1;
    else if (guess < SECRET)
        status = -1;
    return status;
}
...
```



The `assert`*Fact*`()` Methods

- The `assert`*Fact*`()` methods allow different conditions to be checked during testing.
- The `assert`*Fact*`()` methods are defined in the `junit.framework.Assert`, and are inherited by the `junit.framework.TestCase` subclass.
- All methods are `static`, `void` and overloaded.
- The `String msg` is printed if the test condition *fails*.
- In the first two `assertEquals()` methods below, *Type1* can be one of the following: `boolean`, `byte`, `char`, `short`, `int`, `long`, `String` and `Object`.
- In the last two `assertEquals()` methods, *Type2* is either `float` or `double`.
- All occurrences of *Type1* or *Type2* in a parameter list must be of the same type.

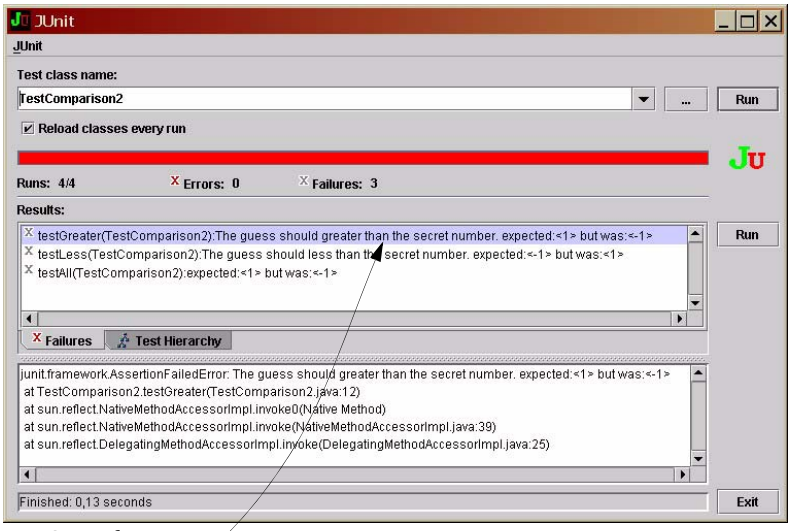
`junit.framework.Assert`	

`assertEquals(`*Type1* `exp, `*Type1* `act)`
`assertEquals(String msg, `*Type1* `exp, `*Type1* `act)`
`assertEquals(`*Type2* `exp, `*Type2* `act, `*Type2* `delta)`
`assertEquals(String msg, `*Type2* `exp, `*Type2* `act, `*Type2* `delta)`
	Compares two values for equality.
	The test passes if the values are equal.
	Floating-point values are compared for equality within a delta.
	Objects are compared for object value equality by calling the `equals()` methods.

junit.framework.Assert	
`assertTrue(boolean condition)` `assertTrue(String msg, boolean condition)`	The test passes if the `boolean` condition expression evaluates to `true`.
`assertFalse(boolean condition)` `assertFalse(String msg, boolean condition)`	The test passes if the `boolean` condition expression evaluates to `false`.
`assertNull(Object obj)` `assertNull(String msg, Object obj)`	The test passes if the reference `obj` is `null`.
`assertNotNull(Object obj)` `assertNotNull(String msg, Object obj)`	The test passes if the reference `obj` is not `null`.
`assertSame(Object exp, Object act)` `assertSame(String msg, Object exp, Object act)`	The test passes if the expression `(exp == act)` is `true`, i.e. the references are *aliases*, denoting the same object.
`assertNotSame(Object exp, Object act)` `assertNotSame(String msg, Object exp, Object act)`	The test passes if the expression `(exp == act)` is `false`, i.e. the references denote different objects.
`fail()` `fail(String msg)`	The current test is forced to fail. See section on exception handling in unit testing.

The `assert`*Fact*() Methods (cont.)

The optional `String` argument in an `assert`*Fact*`()` method should be used to describe the assertion condition, rather than why the assertion condition failed.




```
public class TestComparison2 extends TestCase {
    ...
    public void testGreater() {
        Comparison firstOperand = new Comparison(2004);
        assertEquals("The guess should be greater than the secret number.",
                     1, firstOperand.compare(2010));
    }
   ...
}
```

Using Equality Comparisons for Primitive Values

- The `assertEquals()` methods use the == operator to test the expected primitive value with the actual primitive value for equality.

```
assertEquals(expectedRPM, actualRMP);
assertTrue("Identical revolutions per minute.", expectedRPM == actualRPM);

assertEquals("Returns the same letter.", 'a', str.charAt(0));
assertTrue("Returns the same letter.", 'a' == str.charAt(0));
```

- Floating-point numbers are compared for equality accurate to within a given delta.

```
assertEquals("Atomic Weight",                // Message
             expectedAtomicWeight,           // Expected result
             calculateAtomicWeight(),        // Actual result.
             0.1E-10);                       // Delta
```


Using Equality Comparisons for Objects

- The `assertEquals()` methods use the `equals()` method to test the expected object with the actual object for equality, i.e. the method tests for object value equality.

```
assertEquals(expectedArrivalTimeObj, actualArrivalTimeObj);
assertEquals("Same criminal expected", suspect,
              crimeRegister.matchProfile(suspect));
assertEquals("Should have the same slogan.", // Message
             "Copy once, run everywhere!",   // Expected result
              company.getSlogan());          // Actual result.
```

- The `assertSame()` methods use the == operator to test the expected object with the actual object for equality, i.e. the method tests for object reference equality.

```
assertSame("Should find the same object.", key,
           lookup(keyObject));                    // (1) Passes if aliases.
assertTrue("Should find the same object.",
           key == lookup(keyObject));             // Equivalent to (1).
```

More Examples of `assert`*Fact*() Methods

- Checking a Boolean condition.

```
assertTrue("The set should be empty.", set.getSize() == 0);
assertTrue("Value is non-negative.", actualValue > 0);
assertFalse("Value is non-negative.", actualValue <= 0);
```

- Checking for `null` values.

```
assertNull("No result from the query.", db.doQuery(query));
assertTrue("No result from the query.", db.doQuery(query) == null);

assertNotNull("Lookup should be successful.", db.doQuery(query));
assertTrue("Lookup should be successful.", db.doQuery(query) != null);
assertFalse("Lookup should be successful.", db.doQuery(query) == null);
```

- Causing explicit failure.

```
fail("Cannot proceed.");  // The test always fails.
```


Granularity of Unit Tests

- A unit test should only test conditions that are related to one piece of functionality.
- A unit test fails if an assertFact() method call fails, and the remaining conditions are not checked.
  - If the remaining conditions pertain to unrelated functionality, this functionality will not be tested -- leading to bad test design.
  - Factoring test conditions into appropriate unit tests ensures that these conditions will be tested -- leading to a better test design.
- A unit test should be structured in such a way that if a test condition fails, the remaining conditions will always fail.

Invoking Unit Test Methods

- Each of the unit test methods in a test case is executed as follows:
  - JUnit creates a new instance of the test case for each unit test method.
  - JUnit calls the `setUp()` method in the test case.
  - JUnit calls the unit test method.
  - JUnit calls the `tearDown()` method in the test case.
- Consequence: instance fields in the test case object cannot be used to share state between unit test methods.
- The `setUp()` and `tearDown()` methods can be used to avoid *duplicate code* in the unit test methods.
  - Use the `setUp()` method for duplicate code that creates any resources that each unit test method needs.
  - Use the `tearDown()` method for duplicate code that frees any resources that were used to run each unit test method.
- The constructor of the test case class can also be employed to do the set up.
  - The `setUp()` method is preferred as it provides better documentation of the testing process.
  - The `setUp()` method is called after the test case constructor has been called.

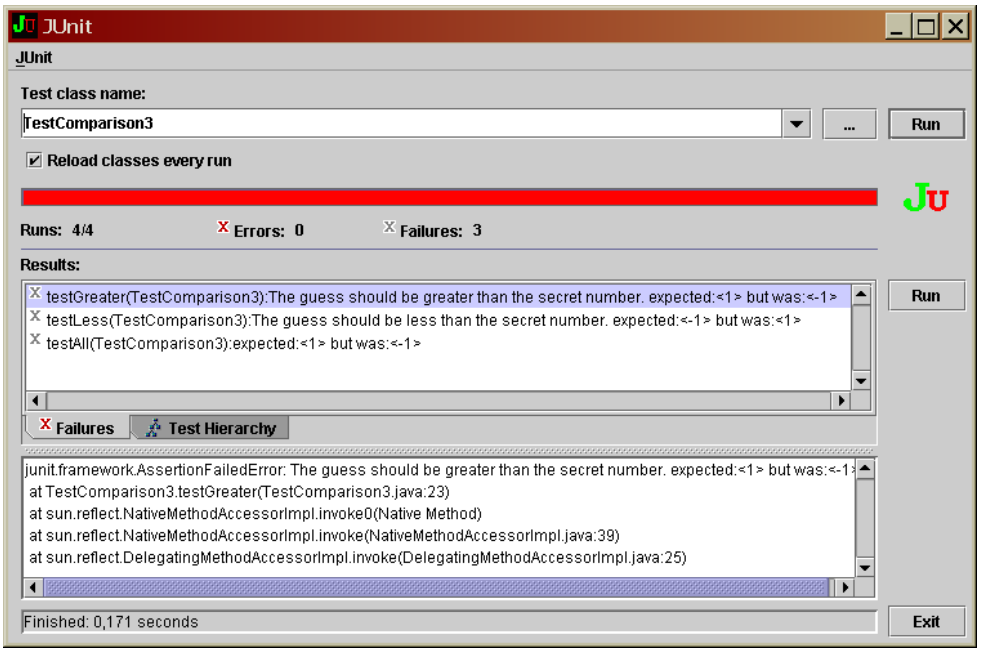

```
/** TestComparison3.java */
import junit.framework.TestCase;
public class TestComparison3 extends TestCase {
    private Comparison firstOperand;
    ...
    public void setUp() {
        System.out.println("Setting up.");
        firstOperand = new Comparison(2004);
    }
    public void tearDown() {
        System.out.println("Tearing down.");
        firstOperand = null;
    }
    public void testEqual() {
        assertEquals("The secret number and guess should be equal.",
                     0, firstOperand.compare(2004));
    }
    public void testGreater() {
        assertEquals("The guess should be greater than the secret number.",
                     1, firstOperand.compare(2010));
    }
    ...
}
```

Tests Using `setUp()` and `tearDown()`

On the console:

```
class TestComparison3:
Setting up.
class TestComparison3:
Tearing down.
class TestComparison3:
Setting up.
class TestComparison3:
Tearing down.
class TestComparison3:
Setting up.
class TestComparison3:
Tearing down.
class TestComparison3:
Setting up.
class TestComparison3:
Tearing down.
```

Corresponds to each unit test method.



Test Suites

- A test suite consists of test cases and other test suites.
- The test cases (and other test suites) in a test suite all run at once.
- JUnit runs the test defined by the `suite()` method in a test case:

```
import junit.framework.TestCase;    // A test case defines multiple unit tests.
import junit.framework.Test;        // Interface for tests that can be run.
import junit.framework.TestSuite;   // A TestSuite is a Composite of Tests.

public class TestComparison4 extends TestCase {
    // ...
    public static Test suite() {
        return new TestSuite(TestComparison4.class);
    }
    // ...
}
```

- The parameter to the `TestSuite` constructor specifies the test case class whose `test`*Something*`()` methods are to be run.
- *Default test suite*: if no `suite()` method is defined in a test case class, reflection is used to locate and run `test`*Something*`()` methods in the test case class.

Creating Test Suites

- Selecting unit tests to run.

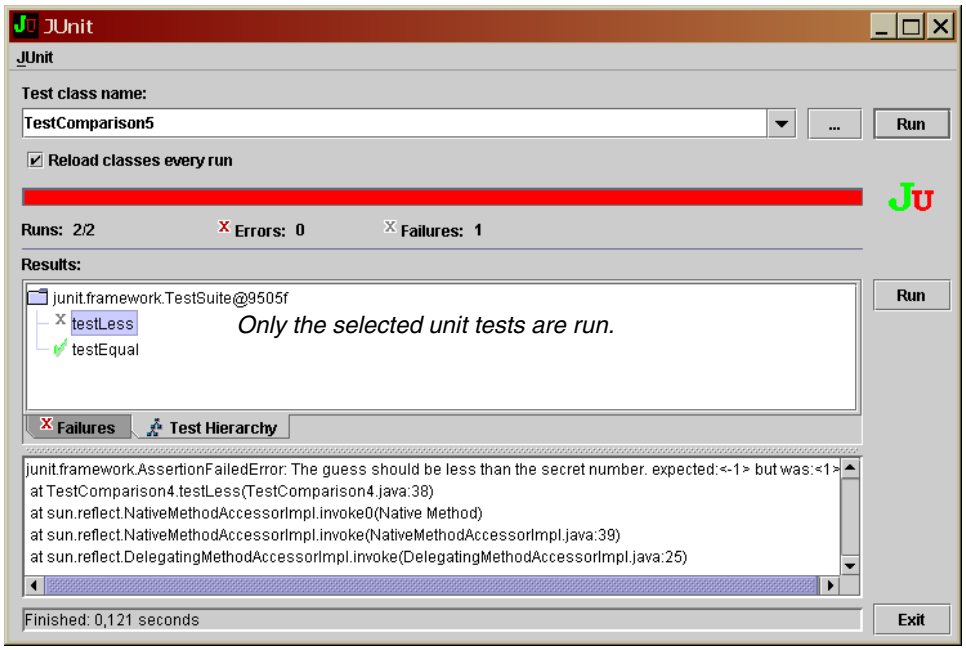
```
public class TestComparison4 extends TestCase {
    public TestComparison4(String testMethodName) { // Mandatory in this case.
        super(testMethodName);
    }
    public void testEqual() { ... }
    public void testLess() { ... }
    ...
}
-----------------------------------------------------------------------
public class TestComparison5 extends TestCase {
    public static Test suite() {
        TestSuite mytestsuite = new TestSuite();
        // Adding 2 individual unit tests
        mytestsuite.addTest(new TestComparison4("testLess"));
        mytestsuite.addTest(new TestComparison3("testEqual"));
        return mytestsuite;
    }
}
```

  - Only tests added to the test suite are run, in the order they were added to the suite.

Running Selected Unit Tests

On the console:

```
class TestComparison4:
Setting up.
class TestComparison4:
Tearing down.
class TestComparison3:
Setting up.
class TestComparison3:
Tearing down.
```



Multiple Test Suites

- Combining multiple suites.

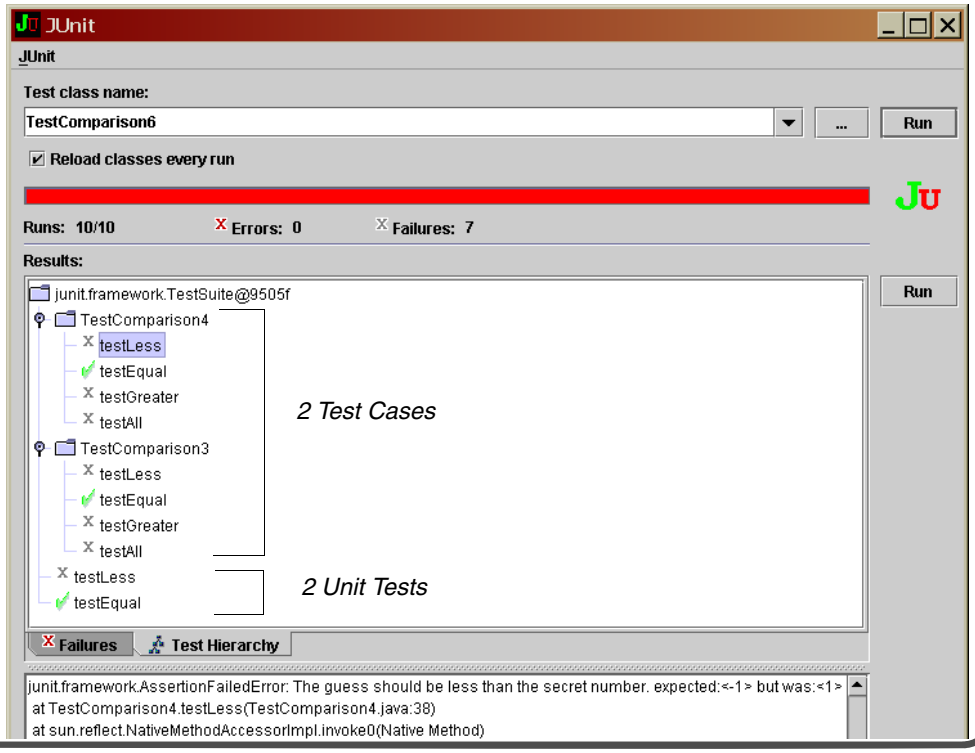
```
public class TestComparison6 extends TestCase {
    public static Test suite() {
        TestSuite mytestsuite = new TestSuite();
        // Adding test suites.
        mytestsuite.addTest(new TestSuite(TestComparison4.class));
        mytestsuite.addTest(new TestSuite(TestComparison3.class));
        // Adding 2 individual unit tests.
        mytestsuite.addTest(new TestComparison4("testLess"));
        mytestsuite.addTest(new TestComparison3("testEqual"));
        return mytestsuite;
    }
}
```

  - Class `TestComparison6` creates a test suite that comprises 2 *test suites* (created from `TestComparison4` and `TestComparison3` test case classes) and 2 selected *unit tests* (one from `TestComparison4` and one from `TestComparison3`).

Running Several Test Suites

On the console:

```
class TestComparison4:
Setting up.
class TestComparison4:
Tearing down.
class TestComparison4:
Setting up.
class TestComparison4:
Tearing down.
class TestComparison4:
Setting up.
class TestComparison4:
Tearing down.
class TestComparison4:
Setting up.
class TestComparison4:
Tearing down.
class TestComparison3:
Setting up.
class TestComparison3:
Tearing down.
class TestComparison3:
Setting up.
class TestComparison3:
Tearing down.
class TestComparison3:
Setting up.
class TestComparison3:
Tearing down.
class TestComparison3:
Setting up.
```



Example II

- Test utility methods in the `ArrayUtil` class.

```
public class ArrayUtil {
    public static void insertionSort(Comparable[] array) { ... }
    public static int binSearch(Comparable[] array, Comparable key) { ... }
    public static void printArray(Comparable[] array, String prompt) { ... }
}
```

- Test case for the sorting method `insertionSort()`.

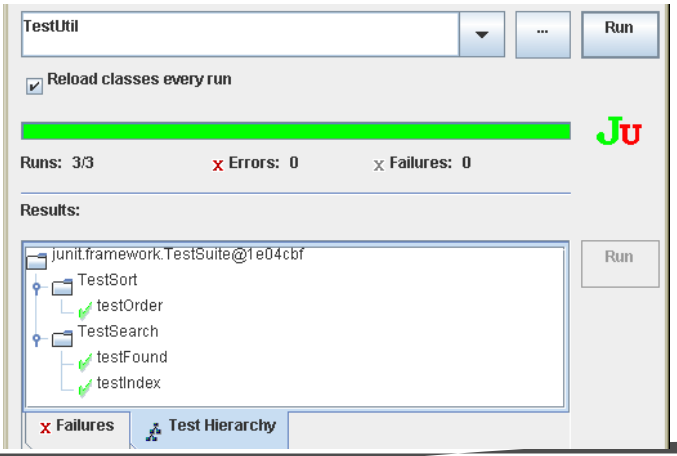
```
public class TestSort extends TestCase {
    Comparable[] array;
    public void setUp() {
        array = new Comparable[] { "This", "is", "not", "so", "difficult" };
    }
    public void testOrder() {
        ArrayUtil.insertionSort(array);
        for (int i = 1; i < array.length; i++) {
            int status = array[i-1].compareTo(array[i]);
            assertTrue("Not sorted. Check index: " + i, status <= 0);
        }
    }
}
```


- Test case for the binary search method `binSearch()`.

```
public class TestSearch extends TestCase {
    Comparable[] array;
    public void setUp() {
        array = new Comparable[] {
                  "Cola 0.5l", "Cola 0.33l", "Pepsi 0.5l",
                  "Solo 0.5l", "Cola 1.0l",  "7Up 0.33l" };
        ArrayUtil.insertionSort(array);
    }
    public void testFound() {
        Comparable key = "Solo 0.5l";
        int index = ArrayUtil.binSearch(array, key);
        assertEquals("Key should in the array: " + key, key, array[index]);
    }
    public void testIndex() {
        Comparable key = "Pepsi 1.0l"; // A key not in the array.
        int index = ArrayUtil.binSearch(array, key);
        assertTrue("Index should be negative: " + index, index < 0);
        key = "Solo 0.5l";             // A key in the array.
        index = ArrayUtil.binSearch(array, key);
        assertTrue("Index should be non-negative: " + index, index >= 0);
    }
}
```

- Test suite for running all the test for class `ArrayUtil`.

```
import junit.framework.*;
public class TestUtil extends TestCase {
    public static Test suite() {
        TestSuite mytestsuite = new TestSuite();
        mytestsuite.addTest(new TestSuite(TestSort.class));
        mytestsuite.addTest(new TestSuite(TestSearch.class));
        return mytestsuite;
    }
}
```

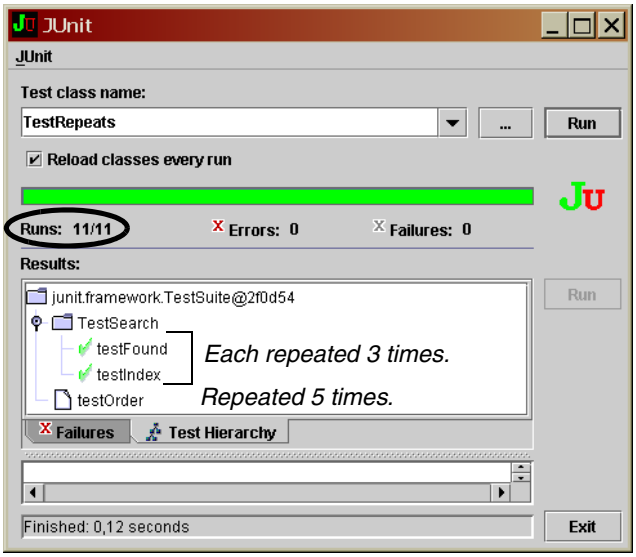


Repeating Tests

- Decorator class `RepeatedTest` allows whole test cases and individual unit tests to be run repeatedly a finite number of times.

```
/** TestRepeats.java */
import junit.framework.*;
import junit.extensions.RepeatedTest; // A Decorator for tests.

public class TestRepeats extends TestCase {
    public static Test suite() {
        TestSuite mytestsuite = new TestSuite();
        // Repeat a whole test case. TestSearch has 2 unit tests, each repeated 3 times.
        mytestsuite.addTest(new RepeatedTest(new TestSuite(TestSearch.class),3));

        // Repeat a single unit test case. In this case,5 times.
        mytestsuite.addTest(new RepeatedTest(new TestSort("testOrder"),5));
        return mytestsuite;
    }
}
```

Repeating Tests (cont.)



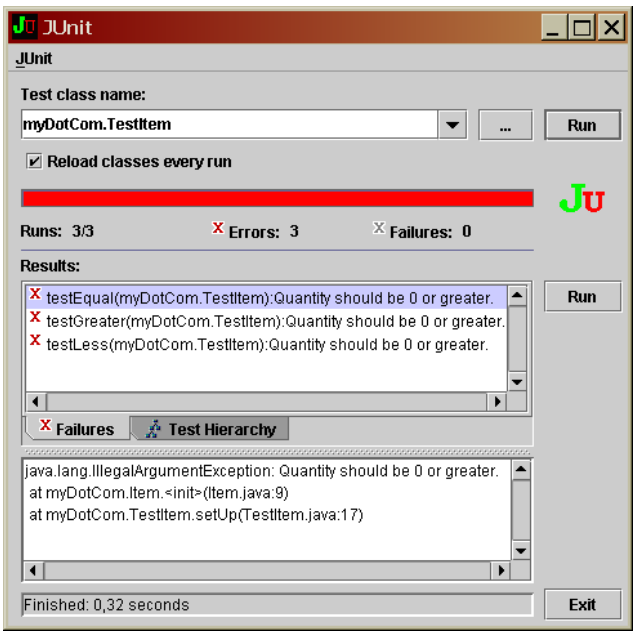
Exception Handling

- Any uncaught exceptions thrown by the code which is being tested will be caught by JUnit and reported.
  - It is superfluous catching these exceptions in the test code.

```
package myDotCom;
public class Item implements Comparable {
    Item(String itemName, int quantity) {
        if (quantity < 0)
            throw new IllegalArgumentException("Quantity should be 0 or greater.");
        this.itemName = itemName;
        this.quantity = quantity;
    }
    ...
}
-----------------------------------------------------------------------
package myDotCom;
public class TestItem extends junit.framework.TestCase {
    public void setUp() {
        item1 = new Item("Cola 0.5l", -1);
    }
    ...
}
```

Reporting of Exceptions

Stack Trace



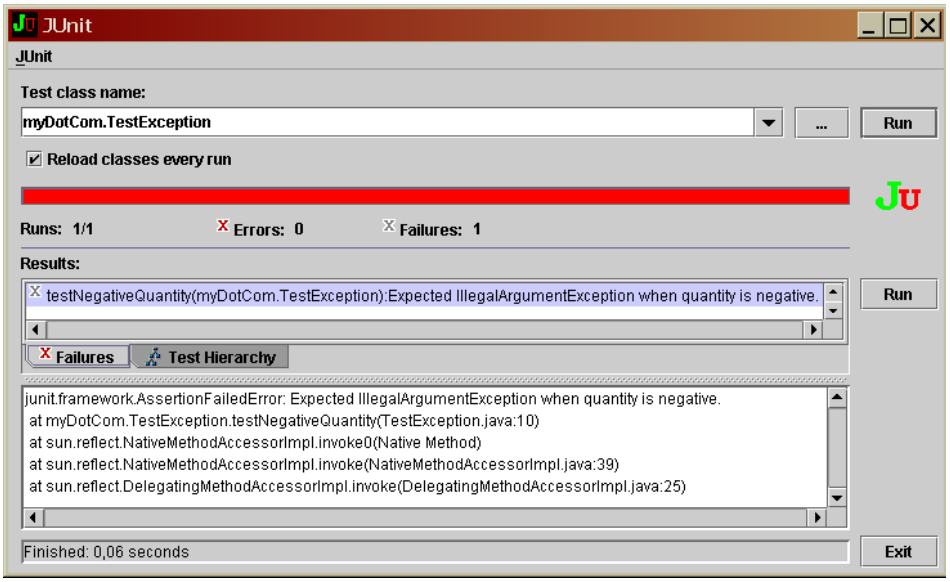
Testing for Exceptions: the `fail()` method

- Testing whether an exception is thrown or not can be done using a `try-catch` block and the `fail()` method.

```
package myDotCom;
public class Item implements Comparable {
    Item(String itemName, int quantity) {
        // Does not check for negative quantity.
        this.itemName = itemName;
        this.quantity = quantity;
    }
    ...
}
-----------------------------------------------
package myDotCom;
public class TestException extends junit.framework.TestCase {
    public void testNegativeQuantity() {
        try {
            Item item = new Item("Cola 0.5l", -1);
            fail("Expected IllegalArgumentException when quantity is negative");
        } catch (IllegalArgumentException iae) {
            // Test passed if the exception was thrown.
        }
    }
}
```

Testing for Exceptions: the `fail()` method (cont.)

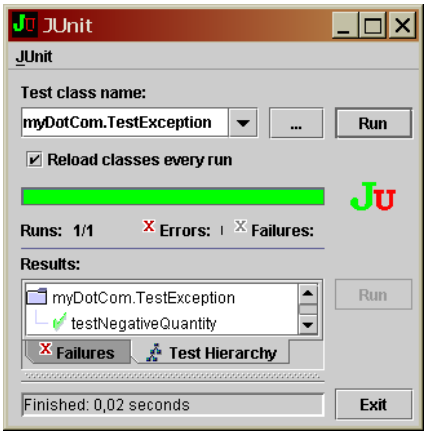
The expected exception was NOT thrown.
The call to the fail() method fails the test.



Testing for Exceptions (cont.)

```
package myDotCom;
public class Item implements Comparable {
    Item(String itemName, int quantity) {
        if (quantity < 0)
            throw new IllegalArgumentException("Quantity should be 0 or greater.");
        this.itemName = itemName;
        this.quantity = quantity;
    }
    ...
}
```

The expected exception was thrown. It was caught and ignored in the unit test method.
The test passes.



Running Tests Concurrently

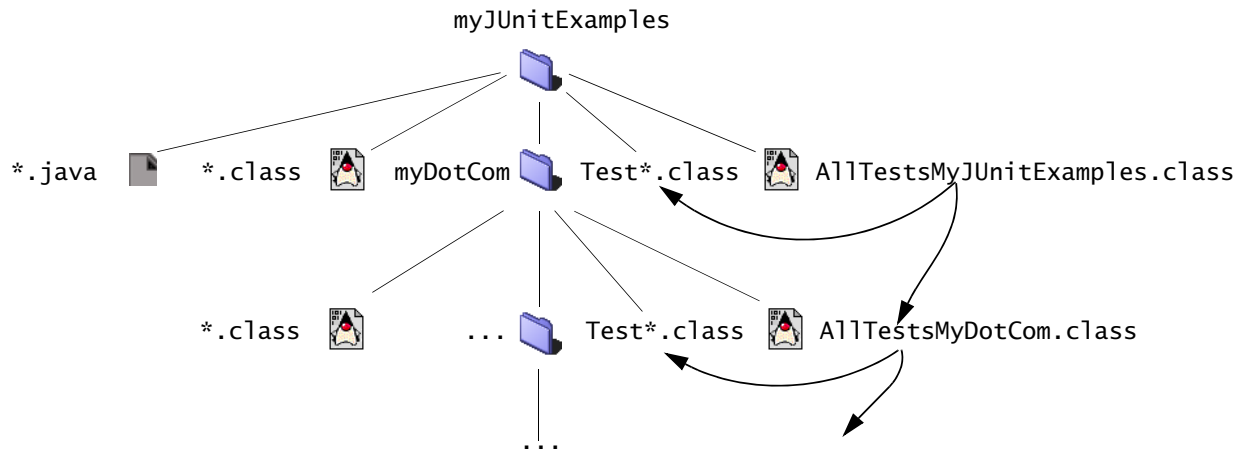
- Tests can be run in their own thread using the `junit.extensions.ActiveTestSuite` class.

```
public class TestConcurrent extends TestCase {
    public static Test suite() {
        TestSuite mytestsuite = new ActiveTestSuite();
       // Repeat a whole test case. TestSearch has 2 unit tests, each repeated 3 times.
        mytestsuite.addTest(new RepeatedTest(new TestSuite(TestSearch.class),3));

        // Repeat a single unit test case. In this case,5 times.
        mytestsuite.addTest(new RepeatedTest(new TestSort("testOrder"),5));
        return mytestsuite;
    }
}
```

- The example above uses 2 threads to run the 2 test suites, *not* 11 threads for the 11 unit tests.
- Useful technique for handling threading problems.

Organizing Tests in Packages

- Aim: run all tests in a package and its subpackages.
- Organization: each package provides a test case that creates a test suite that contains all tests in the current package and its subpackages.

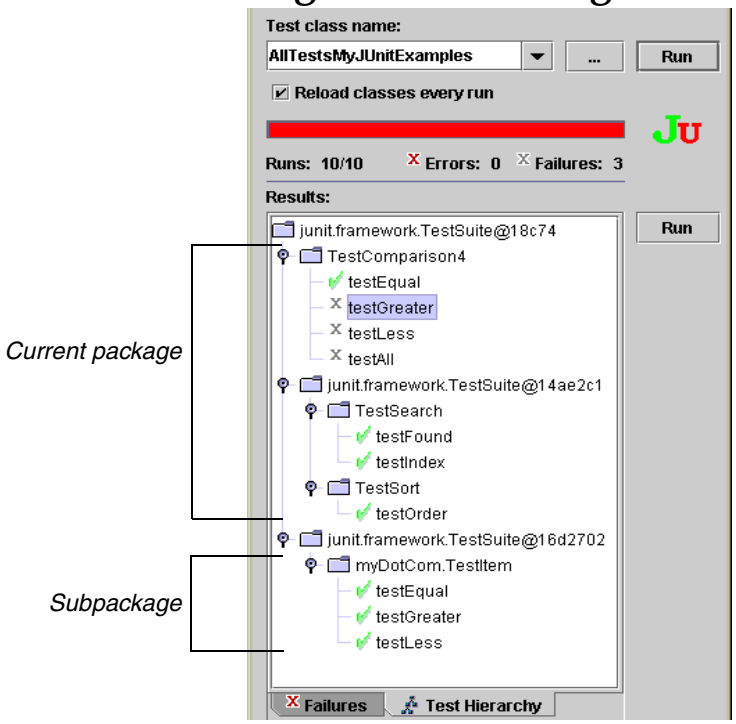


Organizing Tests in Packages (cont.)

```
public class AllTestsMyJUnitExamples extends TestCase {
    public static Test suite() {
        TestSuite mytestsuite = new TestSuite();
        // Add tests from the current package.
        mytestsuite.addTest(new TestSuite(TestComparison4.class));
        mytestsuite.addTest(TestUtil.suite());
        // Add tests from all subpackages.
        mytestsuite.addTest(myDotCom.AllTestsMyDotCom.suite());
        return mytestsuite;
    }
}
```

```
package myDotCom;
import junit.framework.*;
public class AllTestsMyDotCom extends TestCase {
    public static Test suite() {
        TestSuite mytestsuite = new TestSuite();
        mytestsuite.addTest(new TestSuite(myDotCom.TestItem.class));
        return mytestsuite;
    }
}
```


Running Tests in Packages



Running A Unit Test Method Repeatedly

- Purpose: test a method with a wide range of input data.
- Example: test the `equals()` method of a class (`myDotCom.Item`) for the following 6 cases.

```
o1.equals(null)  // false for null comparison
o1.equals(someOtherClassObject)    // false for objects of different classes
o1.equals(o1)    // true for reflexivity
o1.equals(o2)    // (1) true for objects with the same state
o2.equals(o1)    // (2) true for objects with the same state
                 // Both (1) and (2) are true for symmetry of objects with
                 // the same state.
o1.equals(o3)    // false for objects of same class which have different states
```


Procedure for Creating a Test Suite for Testing a Method Repeatedly

1. Specify a test case by subclassing the `TestCase` class.

   ```
   public class TestEquals extends TestCase { ... }
   ```

2. Specify a nested static class to encapsulate the input data for a test and the expected result.

   ```
   private static class TestData {
       Object obj1;
       Object obj2;
       boolean expectedResult;
       public TestData(Object o1, Object o2, boolean expectedResult) {
           this.obj1 = o1;
           this.obj2 = o2;
           this.expectedResult = expectedResult;
       }
   }
   ```

3. Specify the following fields:

   ```
   /** Instance field to distinguish each unique input data. */
   private int testNumber;
   /** Static array with data for each unique input. */
   private static TestData[] testArray;
   ```

4. Specify a constructor for the test case that takes an extra argument.

```
/** A unique test number is passed to the constructor to identify
    the input data for a given test. */
public TestEquals(String testMethodName, int testNumber) {
    super(testMethodName);
    this.testNumber = testNumber;
}
```

5. Implement the actual unit test method that will be run.

```
/** Unit test method called repeatedly for each instance of the test case. */
public void testEquals() {
    // The testNumber identifies the input data for this test.
    TestData td = testArray[this.testNumber];
    // The method which is called.
    boolean result = td.obj1.equals(td.obj2);
    // Condition to check the result of the test.
    assertEquals("Test number " + this.testNumber + ": ",
                 td.expectedResult, result);
}
```


6. Implement the `suite()` method which iterates over the input data, creating an instance of the unit test method for each set of input data to test.

```
/** Creates the test data and the test suite that contains the instances
    of the unit test method with a unique test number. */
public static Test suite() {
    createData();
    TestSuite mytestsuite = new TestSuite();
    for( int i = 0; i < testArray.length; i++) {
        mytestsuite.addTest(new TestEquals("testEquals",i));
    }
    return mytestsuite;
}
```

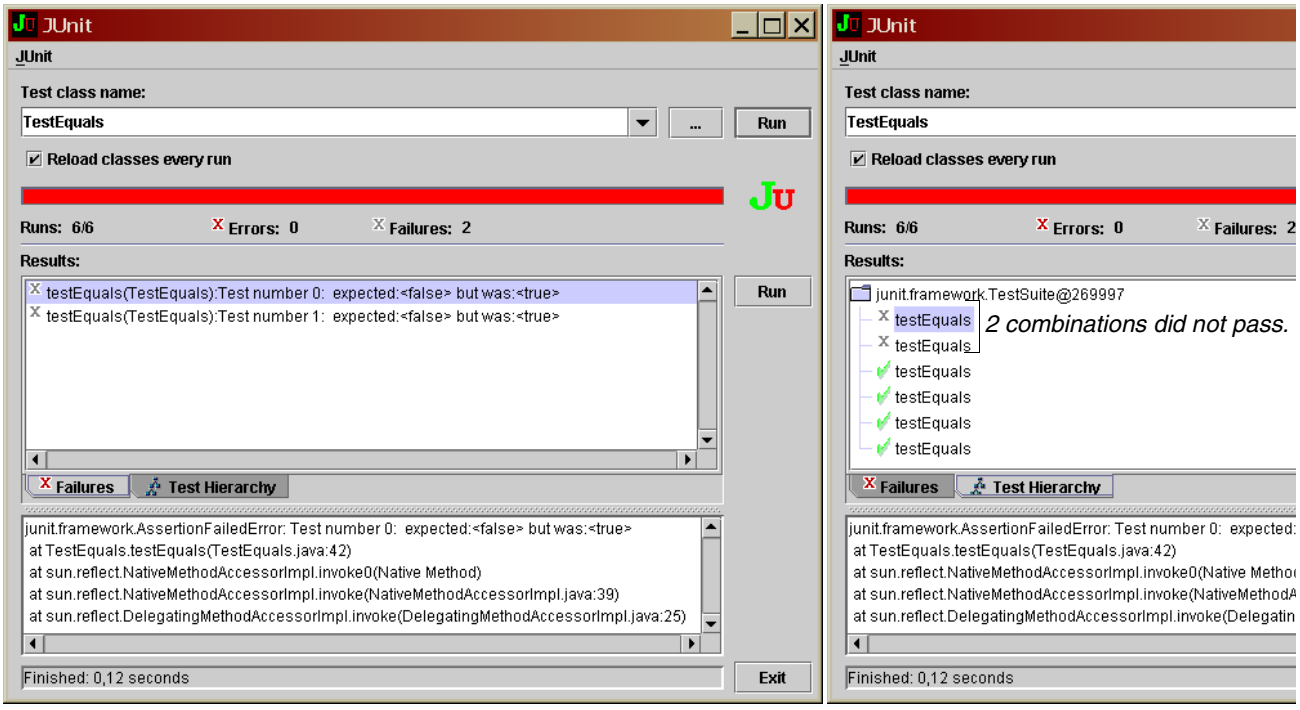
7. Create the input data for each case to test.

```
/** Sets up the input data in the test array for each unit test. */
public static void createData() {
    Object obj1 = new myDotCom.Item("Kola", 10);
    Object obj2 = new myDotCom.Item("Kola", 10);
    Object obj3 = new myDotCom.Item("Kola", 15);
    Object someOtherObject = new Integer(4);
```

```
        testArray = new TestData[] {
            new TestData(obj1, null,           false), // null comparison
            new TestData(obj1, someOtherObject, false), // Not same type
            new TestData(obj1, obj1,           true),  // Reflexive
            new TestData(obj1, obj2,           true),  // (1) Symmetric
            new TestData(obj2, obj1,           true),  // (2) Symmetric
            new TestData(obj1, obj3,           false)  // Same type
        };
    }
```

- The procedure tests each case regardless of whether a test has failed.
- Test input data can be obtained from external sources.

Running the Same Unit Test Method Repeatedly



Encapsulating Common Behavior of Tests

- The common behavior can be encapsulated into a subclass of the TestCase class.
- This subclass can be abstract and can have protected methods that the customized test cases can override.

```
public abstract class CommonTestCase extends TestCase {
    public CommonTestCase(String testMethodName) {
        super(testMethodName);
        initializeStuff();
    }
    protected void initializeStuff() { /*...*/ }
    // ...
}
-------------------------------------------------------------
public class TestCaseSpecific extends CommonTestCase {
    public TestCaseSpecific(String testMethodName) {
        super(testMethodName);
    }
    public void testStuff() { /*...*/ }
    // ...
}
```