JUnit4 - Unit Testing Made Easy

Advanced Topics in Java

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

Version date: 2007-02-03

Overview

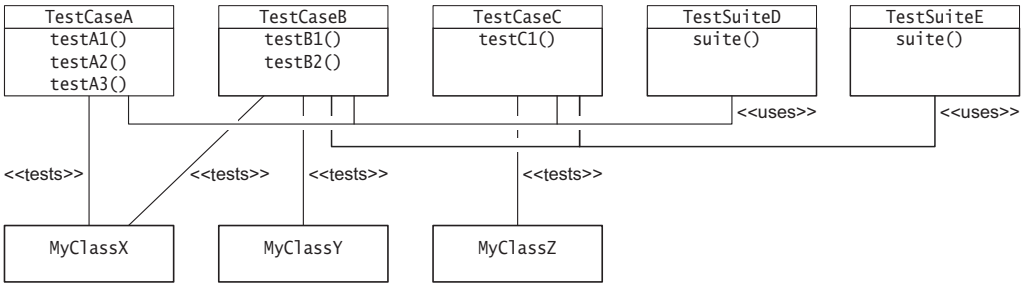
Software Testing

- *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 Java class that declares at least one unit test.
  - A *unit test* is a `public`, `void`, no-parameter method with the annotation `@Test`.
  - A *test suite* is a collection of *test cases*, declared as an empty Java class with the annotations `@RunWith` and `@Suite`.
- 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 (`Junit4.2.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 has the path `C:\Junit4.2`.
- Assign the path `C:\Junit4.2` to the environment variable `JUNIT_HOME`.
- Add the path of the `junit-4.2.jar` file in the `JUNIT_HOME` directory to your `CLASSPATH` environment variable, i.e. the path `C:\JUNIT_HOME\junit-4.2.jar` should be added to the `CLASSPATH` variable.

Writing Simple Unit Tests with JUnit

Procedure for writing a test case:

1. Declare a Java class that will function as a test case.
   - 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 methods in the test case class.
   - The declaration of the unit test method must have the annotation `@Test`.
   - A unit test method must be `public`, `void`, and have no parameters.
   - The JUnit framework requires that a test case has at least one unit test method.
3. Call `assert`*Fact*`()` methods in each unit test method to perform the actual testing.
   - Various overloaded `assert`*Fact*`()` methods are declared in the `org.junit.Assert` class to test different conditions.
   - Typically, the `assertEquals()` method is used to assert that its two arguments are equal.
   - *Statically import* `assertFact()` methods from the `org.junit.Assert` class for convenience.
4. Can also use `assert` statements to perform the tests.

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:

```
@Test public void testEqual()   { ... }
@Test public void testGreater() { ... }
@Test public void testLess()    { ... }
@Test 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 `jar` file with the JUnit framework is in the classpath of the compiler when compiling test case classes.

```
/** TestComparison.java */
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class TestComparison {           // (0) Test case
    @Test public void testEqual() {                          // (1) Unit test
        Comparison firstOperand = new Comparison(2004);
        assertEquals(0, firstOperand.compare(2004));
    }
    @Test public void testGreater() {                        // (2) Unit test
        Comparison firstOperand = new Comparison(2004);
        assertEquals(1, firstOperand.compare(2010));
    }
    @Test public void testLess() {                           // (3) Unit test
        Comparison firstOperand = new Comparison(2004);
        assertEquals(-1, firstOperand.compare(2000));
    }
    @Test 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));
    }
}
```

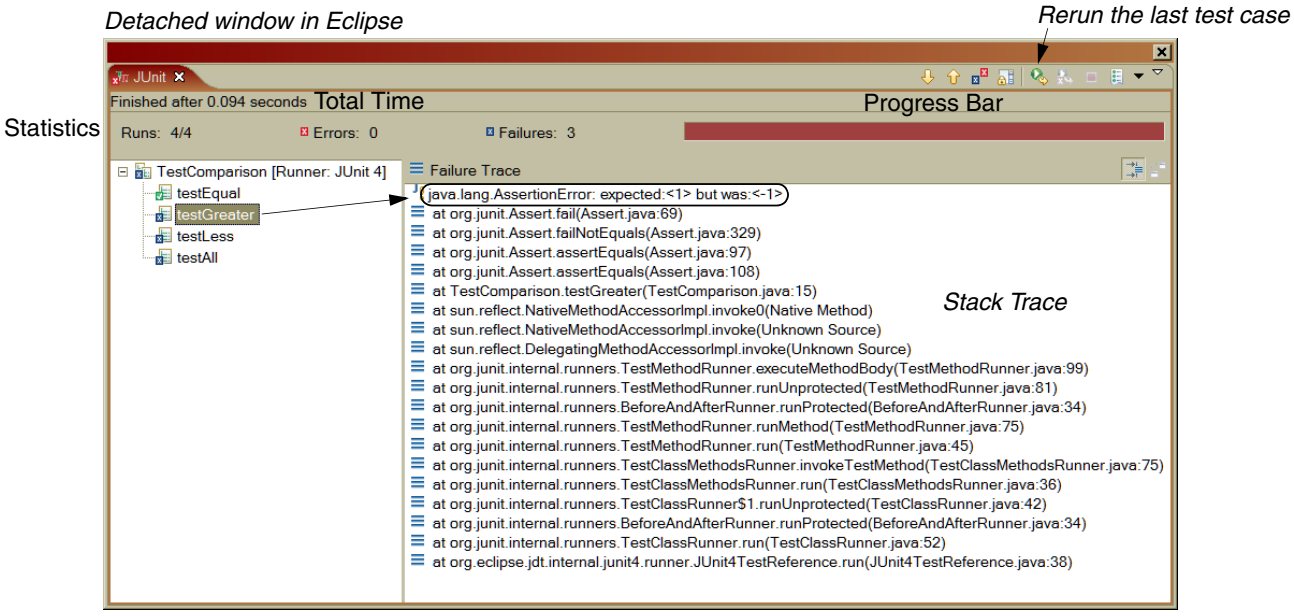

Running Unit Tests: Text mode

```
>java -ea org.junit.runner.JUnitCore TestComparison
JUnit version 4.2           <=========== JUnit version
..E.E.E                     <=========== Progress line
Time: 0.031                 <=========== Total time
There were 3 failures:      <=========== Failures
1) testGreater(TestComparison)
java.lang.AssertionError: expected:<1> but was:<-1>
        at org.junit.Assert.fail(Assert.java:69)
        ...
2) testLess(TestComparison)
java.lang.AssertionError: expected:<-1> but was:<1>
        at org.junit.Assert.fail(Assert.java:69)
        ...
3) testAll(TestComparison)
java.lang.AssertionError: expected:<1> but was:<-1>
        at org.junit.Assert.fail(Assert.java:69)
        ...

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

Test Failures

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

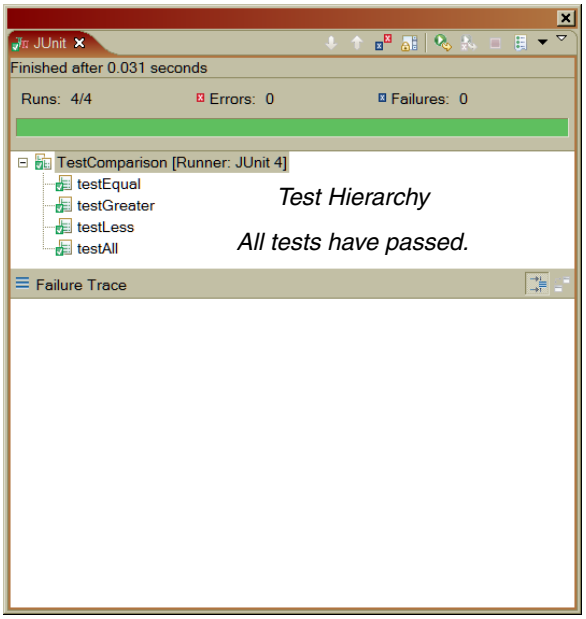
Running Unit Tests: GUI mode in Eclipse



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 static `assert`*Fact*() Methods

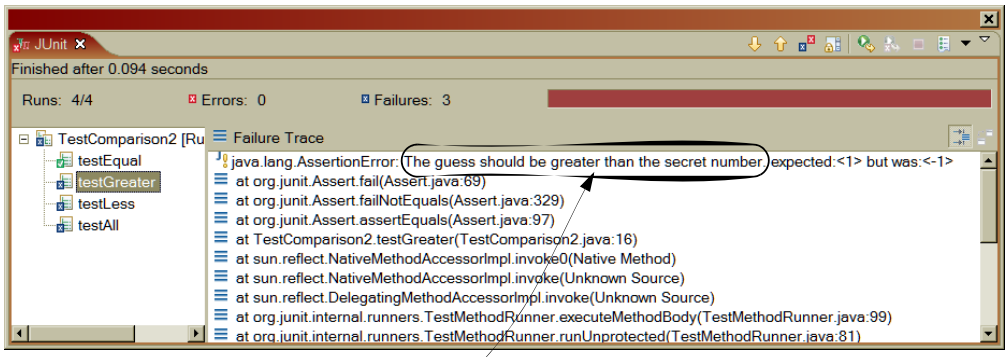
- The `assert`*Fact*`()` methods allow different conditions to be checked during testing.
- The `assert`*Fact*`()` methods are defined in the `org.junit.Assert`.
- All methods are `static`, `void` and overloaded.
- The `String msg` is printed if the test condition *fails*.

`org.junit.Assert`	
`assertEquals(Object exp, Object act)` `assertEquals(String msg, Object exp, Object act)`	Compares two values for equality. The test passes if the values are equal. Objects are compared for object value equality by calling the `equals()` methods. Arrays are compared *element-wise*. Note that arrays with primitive values cannot be compared.
`assertEquals(Object[] exp, Object[] act)` `assertEquals(String msg, Object[] exp, Object[] act)`	
`assertEquals(`*Type1* `exp, `*Type1* `act, `*Type1* `delta)` `assertEquals(String msg, `*Type1* `exp, `*Type1* `act, `*Type1* `delta)`	*Type1* is either `float` or `double`. Floating-point values are compared for equality within a delta.
`assertTrue(boolean condition)` `assertTrue(String msg, boolean condition)`	The test passes if the `boolean` condition expression evaluates to `true`.

org.junit.Assert	
`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 {
    ...
    @Test 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 `equals()` method to test the *boxed* expected primitive value with the *boxed* actual primitive value for equality.

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

assertEquals("Returns the same letter.", 'a', str.charAt(0)); // Boxed
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.
assertEquals("Guest list not correct.",      // Message
             expectedArrayOfGuests,          // Expected array
             fetch.actualArrayOfGuests());   // Actual array.
```

- 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 unit 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 `assert`*Fact*`()` method call (or an `assert` statement) 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 method(s) with the annotation `@Before` in the test case.
  - JUnit calls the unit test method.
  - JUnit calls method(s) with the annotation `@After` in the test case.
- Consequence: instance fields in the test case object cannot be used to share state between unit test methods.
- The `@Before` and `@After` methods can be used to avoid *duplicate code* in the unit test methods.
  - Use the `@Before` methods for duplicate code that creates any resources that each unit test method needs.
  - Use the `@After` methods 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 `@Before` methods are preferred as they provides better documentation of the testing process.
  - The `@Before` methods are called after the test case constructor has been called.

The `@Before` and `@After` Annotations

- A `@Before` or `@After` method must be `public` and non-`static`.
- A test case class can have multiple `@Before` or `@After` methods.
  - Order of execution of the `@Before` methods (and analogously that of the `@After` methods) is unspecified in the test case class.
- A test case class can inherit `@Before` and `@After` methods from its superclasses.
- `@Before` methods inherited from the superclasses are run *before* `@Before` methods in the test case.
- `@After` methods inherited from the superclasses are run *after* `@After` methods in the test case.
- *Overridden* `@Before` or `@After` methods from the superclasses are *not* run.
- All `@After` methods are guaranteed to be executed regardless of whether a `@Before` or a `@Test` method throws an exception.

```java
/** TestComparison3.java */
import static org.junit.Assert.assertEquals;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
public class TestComparison3 {
  private Comparison firstOperand;

  @Before public void initialize() {
    System.out.println(this.getClass() + ": Initializing.");
    firstOperand = new Comparison(2004);
  }

  @After public void cleanUp() {
    System.out.println(this.getClass() + ": Cleaning up.");
    firstOperand = null;
  }

  @Test public void testEqual() {
    assert 0 == firstOperand.compare(2004)              // Using assert statement
    : "The secret number and guess should be equal.";
  }
```



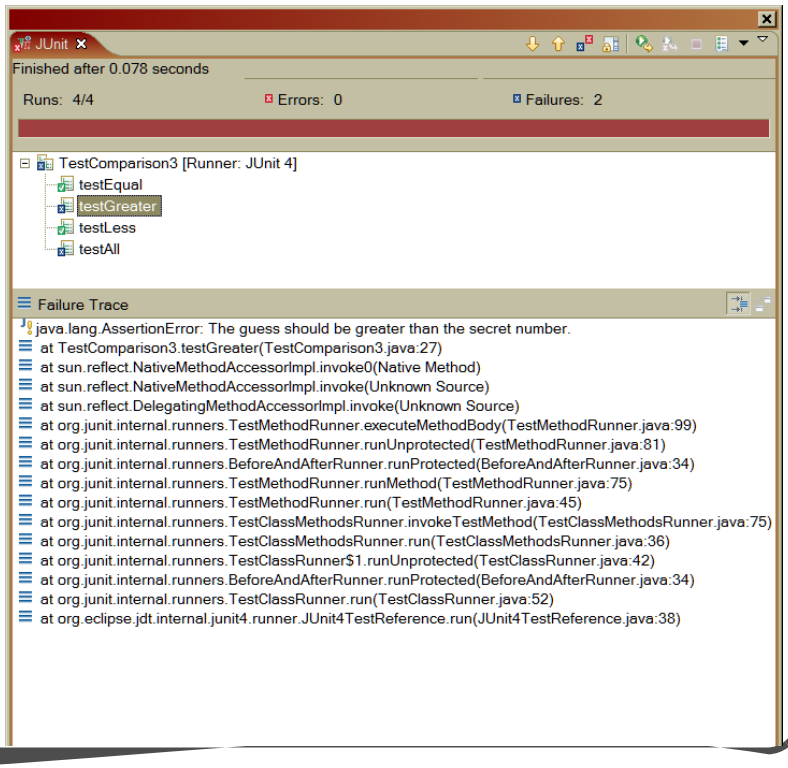
```java
  @Test public void testGreater() {
    assert 1 == firstOperand.compare(2010)
    : "The guess should be greater than the secret number.";
  }

  @Test public void testLess() {
    assert -1 == firstOperand.compare(2000)
    : "The guess should be less than the secret number.";
  }

  @Test public void testAll() {
    assertEquals( 0, firstOperand.compare(2004));
    assertEquals( 1, firstOperand.compare(2010));
    assertEquals(-1, firstOperand.compare(2000));
  }
}
```

Using `@Before` and `@After` Annotations

On the console:

```
class TestComparison3: Initializing.
class TestComparison3: Cleaning up.
class TestComparison3: Initializing.
class TestComparison3: Cleaning up.
class TestComparison3: Initializing.
class TestComparison3: Cleaning up.
class TestComparison3: Initializing.
class TestComparison3: Cleaning up.
```

Corresponds to each unit test method.



The `@BeforeClass` and `@AfterClass` Annotations

- The `@BeforeClass` method and the `@AfterClass` method must be `public` and `static` in a test case.
- The `@BeforeClass` method is useful when resources need to be set up *once and for all* for all unit methods, and the `@AfterClass` method is useful when resources need to be released when all the unit tests have run.
  - The `@BeforeClass` method is run before any unit test is run, and the `@AfterClass` method is run when all the unit tests have run, i.e. they run only once.
- A test case class can inherit `@BeforeClass` and `@AfterClass` methods from its superclasses.
- `@BeforeClass` methods inherited from the superclasses are run *before* the `@BeforeClass` method in the test case.
- `@AfterClass` methods inherited from the superclasses are run *after* the `@AfterClass` method in the test case.
- *Hidden* `@BeforeClass` or `@AfterClass` methods from the superclasses are *not* run.
- All `@AfterClass` methods are guaranteed to be executed regardless of whether a `@BeforeClass` or a `@Test` method throws an exception.

```
public class TestComparison3a {

  private Comparison firstOperand;

  @BeforeClass public static void initializeAll() {
    System.out.println(TestComparison3a.class  +
                       ": Initializing once and for all.");
  }

  @AfterClass public static void cleanUpAll() {
    System.out.println(TestComparison3a.class +
                       ": Cleaning up once and for all.");
  }
  // ...
}
```


The `@Ignore` Annotation

- The `@Ignore` annotation, with optional `String` parameter, can be used to skip a unit test, i.e. the unit test is not run.

```
public class TestComparison3a {

  private Comparison firstOperand;

  /...
  @Ignore("Not ready to run.")
  @Test public void testAll() {
    assertEquals( 0, firstOperand.compare(2004));
    assertEquals( 1, firstOperand.compare(2010));
    assertEquals(-1, firstOperand.compare(2000));
  }
}
```

Using `@BeforeClass`, `@AfterClass` and `@Ignore` Annotations

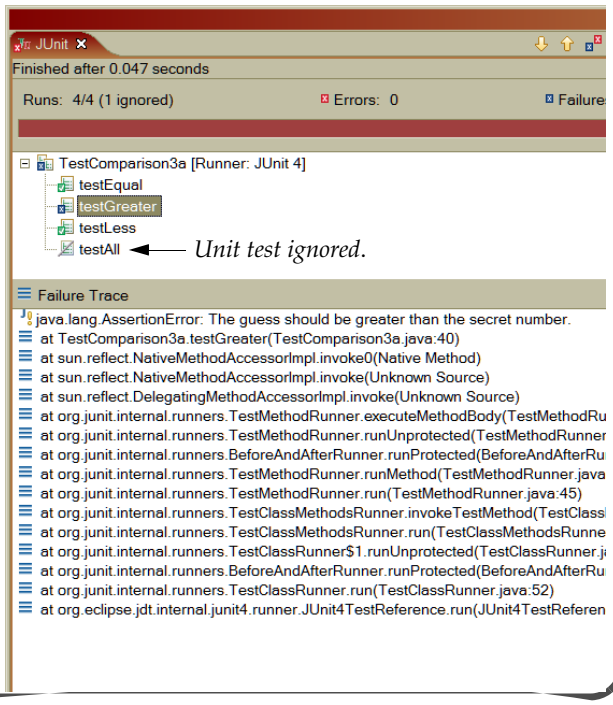
Corresponds to `@BeforeClass` *method.*

On the console:

```
class TestComparison3a: Initializing once and for all
class TestComparison3a: Initializing.
class TestComparison3a: Cleaning up.
class TestComparison3a: Initializing.
class TestComparison3a: Cleaning up.
class TestComparison3a: Initializing.
class TestComparison3a: Cleaning up.
class TestComparison3a: Cleaning up once and for all
```

Only 3 unit tests were run.

Corresponds to `@AfterClass` *method.*



Test Suites

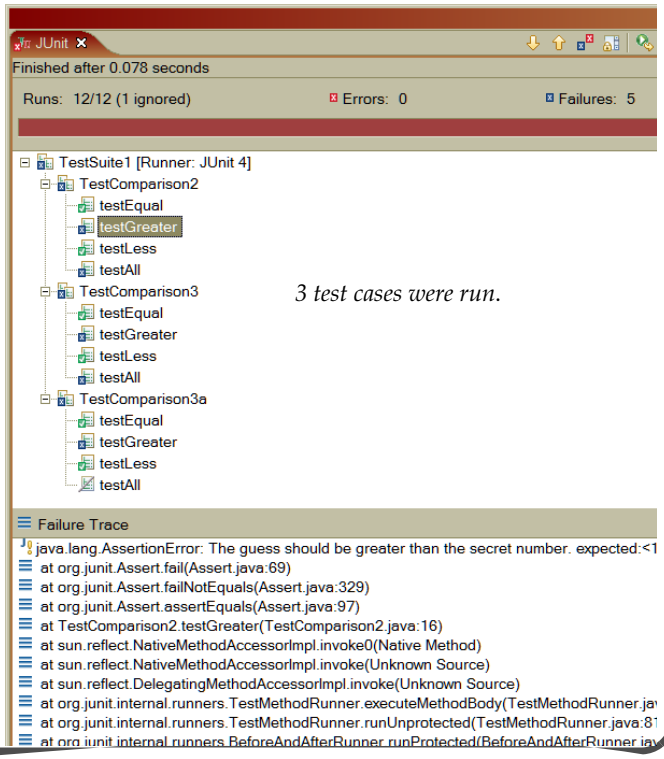
- A test suite consists of test cases and other test suites.
- The test cases (and other test suites) in a test suite all run at once.
- The `@RunWith` annotation is used to specify that the `org.junit.runner.Suite` should be used to run the suite.
- The `@Suite.SuiteClass` annotation is used to specify the test cases (and test suites) to run.

```
// The annotation types
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)                  // The runner to use
@Suite.SuiteClasses(                   // Test cases to run
  {TestComparison2.class, TestComparison3.class, TestComparison3a.class}
)
public class TestSuite1 { }    // Empty class
```

Running a Test Suite

On the console:

```
class TestComparison3: Initializing.
class TestComparison3: Cleaning up.
class TestComparison3: Initializing.
class TestComparison3: Cleaning up.
class TestComparison3: Initializing.
class TestComparison3: Cleaning up.
class TestComparison3a: Initializing once and for
all.
class TestComparison3a: Initializing.
class TestComparison3a: Cleaning up.
class TestComparison3a: Initializing.
class TestComparison3a: Cleaning up.
class TestComparison3a: Initializing.
class TestComparison3a: Cleaning up.
class TestComparison3a: Cleaning up once and for
all.
```



3 test cases were run.

Creating Multiple Test Suites

- Combining test cases and suites.

```
/** TestSuite2.java */
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses(
    {TestComparison2.class, TestComparison3.class, TestComparison3a.class,
     TestSuite1.class}  // Added a test suite.
)
public class TestSuite2 {}
```

  - Class `TestSuite2` creates a test suite that comprises 3 test cases (`TestComparison2`, `TestComparison3` and `TestComparison3a`) and 1 *test suite* (`TestSuite1`), altogether 24 unit tests.

Example II

- Test utility methods in the `ArrayUtil` class.

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

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

```
// File: TestSort.java
import static org.junit.Assert.assertTrue;

import org.junit.Before;
import org.junit.Test;

public class TestSort {

  private String[] array;

  @Before public void setUp() {
    array = new String[] { "This", "is", "not", "so", "difficult" };
  }
```



```
  @Test public void testOrder() { // array[0] <= array[] <= ... <= array[n-1]
    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);
    }
  }
}
```

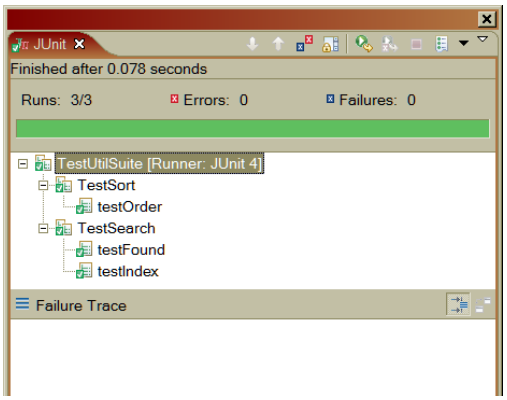
- Test case for the binary search method `binSearch()`.

```
// File: TestSearch.java
// ...
public class TestSearch {
  String[] array;

  @Before public void setUp() {
    array = new String[] {
        "Cola 0.5l", "Cola 0.33l", "Pepsi 0.5l",
        "Solo 0.5l", "Cola 1.0l",  "7Up 0.33l"
    };
    ArrayUtil.insertionSort(array);
  }
```

```
  @Test public void testFound() {
    String key = "Solo 0.5l";
    int index = ArrayUtil.binSearch(array, key);
    assertEquals("Key should in the array: " + key, key, array[index]);
  }
  @Test public void testIndex() {
    String 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`.

```
/** TestUtilSuite.java */
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({TestSort.class, TestSearch.class})
public class TestUtilSuite { }
```



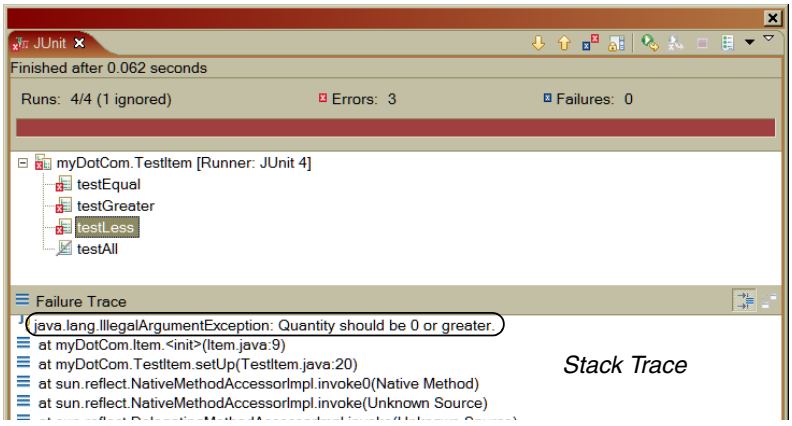
Exception Handling

- 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> {
    Item(String itemName, int quantity) {
        if (quantity < 0)
            throw new IllegalArgumentException("Quantity should be 0 or greater.");
        this.itemName = itemName;
        this.quantity = quantity;
    }
    ...
}
```


Reporting of Exceptions

```
package myDotCom;
public class TestItem {
    @Before public void setUp() {
        item1 = new Item("Cola 0.5l", -1);
    }
    ...
}
```



Stack Trace

Testing for Exceptions: the `fail()` method

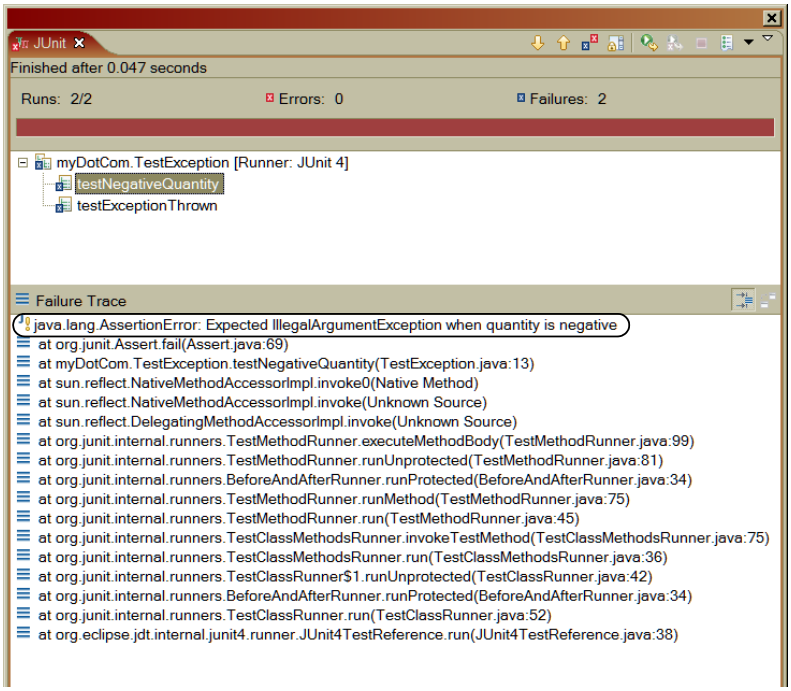
- Testing whether an exception is thrown or not can be done using a `try-catch` block and the `fail()` method.

```
package myDotCom;
public class Item implements Comparable<Item> {
    Item(String itemName, int quantity) {
        // Does not check for negative quantity.
        this.itemName = itemName;
        this.quantity = quantity;
    }
    ...
}
-----------------------------------------------
package myDotCom;
public class TestException {
    @Test 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.)

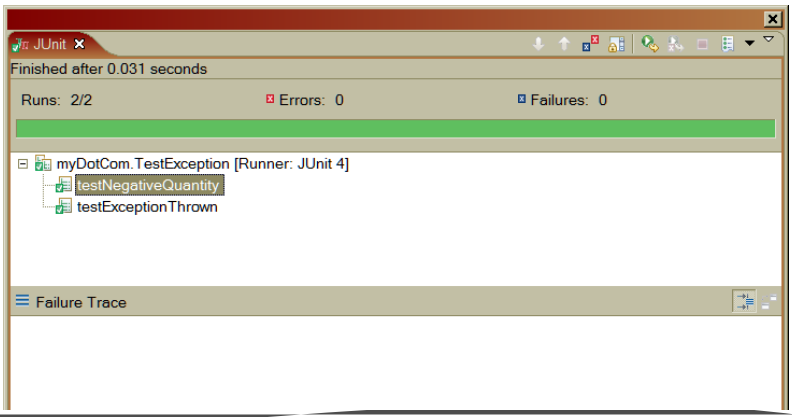
The expected exception was NOT thrown.

The call to the fail() method fails the test.



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

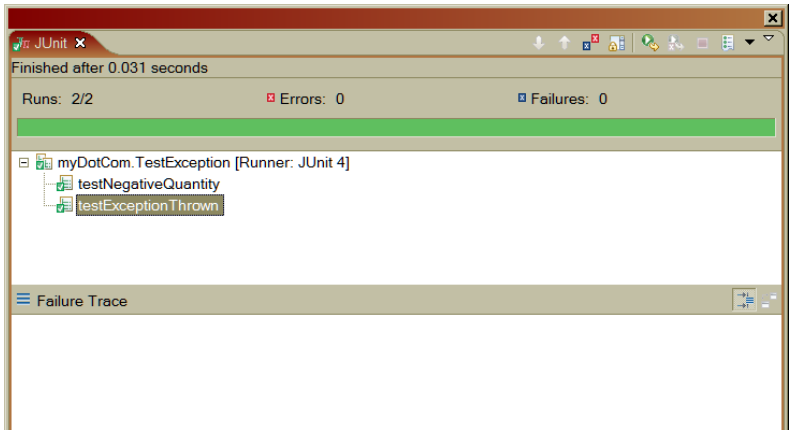
```
package myDotCom;
public class Item implements Comparable<Item> {
    Item(String itemName, int quantity) {
        if (quantity < 0)
            throw new IllegalArgumentException("Quantity should be 0 or greater.");
        this.itemName = itemName;
        this.quantity = quantity;
    }
    ...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



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

Testing for Exceptions: `@Test(expected = XException.class)`

```
public class TestException {
  // ...
  @Test(expected = IllegalArgumentException.class)
  public void testExceptionThrown() {
    Item item = new Item("Cola 0.5l", -1);
  }
}
```



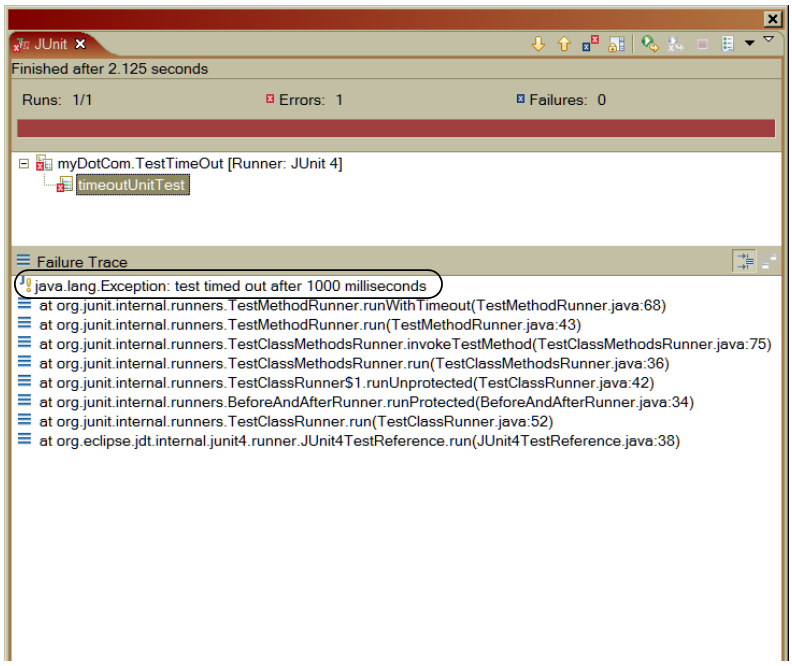
The expected exception was thrown.
The test passes.

Timeout Tests: `@Test(timeout = nnnn)`

- Allow a timeout value (in milliseconds) to be specified for a unit test.
- If no result is returned within the specified time, the unit test fails.

```
package myDotCom;
public class Item implements Comparable<Item> {
  // ..
  public void artificial() { // To demonstrate timeout of unit tests.
    for(;;);
  }
}
-------------------------------------------
/** TestTimeOut.java */
package myDotCom;
import org.junit.Test;
public class TestTimeOut {
  @Test(timeout = 1000)
  public void timeoutUnitTest() {
    Item item = new Item("Cola 0.5l", 10);
    item.artificial();
  }
}
```


Timeout Tests (cont.)



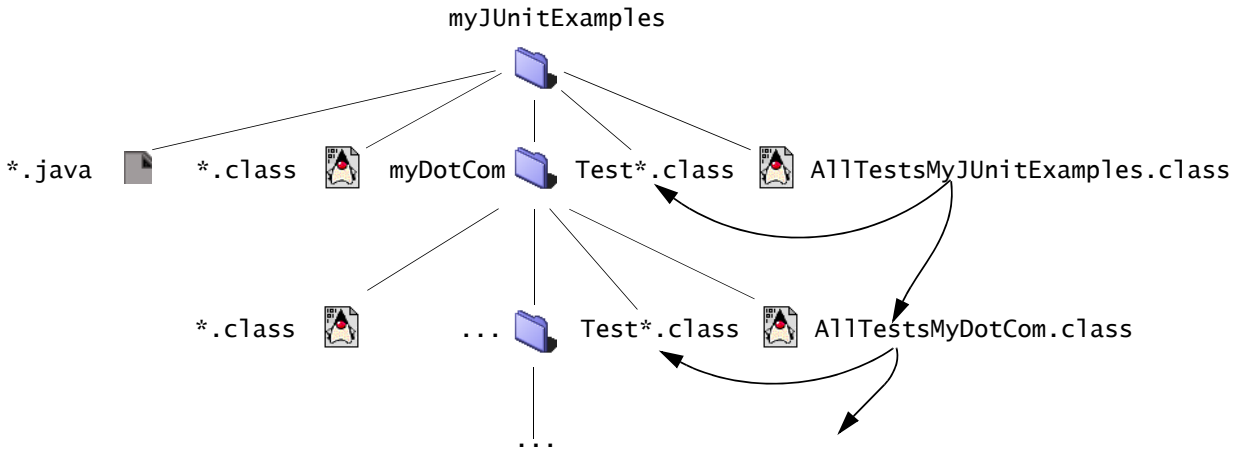
Encapsulating Common Behavior of Tests

- The common behavior of tests can be encapsulated into an abstract class.
- This abstract class can have methods that the customized test cases can override.

```
public abstract class CommonTestCases {
    @Before public void initializeStuff() { /*...*/ }
    @Test public void test1() { /* ... */ }
    // ...
}
-------------------------------------------------------------
public class SpecificTestCase extends CommonTestCases {
    @Test public void test1() { /* ... */ }
    @Test public void testStuff() { /*...*/ }
    // ...
}
```


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.)

```
/** AllTestsMyJUnitExamples.java */

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

@RunWith(Suite.class)
@Suite.SuiteClasses({TestSuite1.class,      // Tests in this package
                     TestUtilSuite.class,   // Tests in this package
       myDotCom.AllTestsMyDotCom.class})    // Tests in myDotCom package
public class AllTestsMyJUnitExamples { }
-----------------------------------------------------------------------
/** AllTestsMyDotCom.java */

package myDotCom;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({TestException.class,   // Tests in this package.
                     TestItem.class})       // Tests in this package.
public class AllTestsMyDotCom  { }
```


Running Tests in Packages

Parameterized Tests: Running Unit Tests 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(someOtherObject)    // 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. Declare a test case with the `@RunWith` annotation as follows:

```
@RunWith(Parameterized.class)
public class TestEquals { ... }
```

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

```
private static class TestData<T1, T2> {
  T1 obj1;
  T2 obj2;
  boolean expectedResult;

  public TestData(T1 o1, T2 o2, boolean expectedResult) {
    this.obj1 = o1;
    this.obj2 = o2;
    this.expectedResult = expectedResult;
  }
}
```

3. Specify the following field for input data to a unit test:

```
private TestData<?,?> td;
```

4. Specify a constructor that receives input data for each unit test run.

```
public TestEquals(TestData<?,?> td) {
  this.td = td;
}
```

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

```
@Test public void testEquals() {
  // Do the test.
  boolean result = td.obj1.equals(td.obj2);
  // Condition to check the result of the test.
  assertEquals("Test failed.", td.expectedResult, result);
}
```

6. Create the input data for each case instance to test.
   – The method is annotated with the `@Parameters` annotation.

```
/** Sets up the input data for the unit test, i.e. Collection of Arrays. */
@Parameters
public static List<TestData<?,?>[]> createData() {
  Item obj1 = new Item("Kola", 10);
  Item obj2 = new Item("Kola", 10);
  Item obj3 = new Item("Kola", 15);
  Integer someOtherObj = new Integer(4);
```



```
  TestData<?,?>[][] testArray = {
    { new TestData<Item, Item>(obj1, null, false) },      // null comparison
   { new TestData<Item, Integer>(obj1, someOtherObj, false) },// Not same type
    { new TestData<Item, Item>(obj1, obj1, true) },  // Reflexive
    { new TestData<Item, Item>(obj1, obj2, true) },  // (1) Symmetric
    { new TestData<Item, Item>(obj2, obj1, true) },  // (2) Symmetric
    { new TestData<Item, Item>(obj1, obj3, false) }  // Same type
  };
  return Arrays.asList(testArray);
}
```

- The procedure tests each case regardless of whether a test has failed.
- Test input data can be obtained from external sources, but must be converted to a collection of arrays.
- *Each* unit test method will be repeated as many times as the number of elements in the input collection of arrays.

  See also the simplified version in file `TestEqualsSimplified.java`.

Running the Same Unit Test Method Repeatedly