

# Unit Testing with JUnit

Subhendu Ghatak

subhendu.ghatak@tcs.com

Subodh Bandhawakar

subodh.b@tcs.com

# What We Will Learn Today

1. What is unit testing and why do it
2. Writing and running test cases with JUnit
3. Test fixtures
4. Test suites
5. Continuous build and test with Ant
6. Checking test quality with code coverage and error injection

# What is Unit Testing?

Test of *functionality* of *small portions* of code in *isolation* from the *full system* to ensure that it meets the *developer's* expectation.

# Why do Unit Testing?

1. If code units work, integration is largely error free
2. Thorough unit testing reduces the need to use a debugger
3. Check code coverage at the unit testing phase itself.
4. Builds confidence and reduces panic as project progresses
5. Passing the unit tests at each build becomes an obsession – so much so, that we may not go home till all applicable tests pass.

# Writing Test Cases with JUnit

- Download and install the latest version of JUnit from [www.junit.org](http://www.junit.org)
- JUnit 4.1 is brought to us by Kent Beck, Erich Gamma, and David Saff.
- JUnit provides the framework to
  - Write tests
  - Run the tests and report results
  - Perform pre and post test operations
  - Group the tests into test suites

# The Classes to be Tested

- Two classes will be tested: Item and Inventory
- Class Item represents a merchandise with four attributes – name, code, manufacturer and quantity.
- The methods of the class Item are:
  - public String getName( )
  - public String getCode( )
  - public String getManufacturer( )
  - public int getQuantity( )

# The Inventory Class

Class Inventory is a collection of items – its methods are:

- `public void addItem( Item item)`
- `public void addItemFromFile(String fileName) throws Exception`
- `public Vector.getItems( String manufacturer)`
- `public Item getItem(String code)`
- `public void removeItem( String code) throws Exception`
- `public void clear( ) // remove all items from the inventory`
- `public int size( ) // get the number of items in the inventory`

# Writing a test: TestInventory1

- TestInventory1 is a test class with a single test method `testClear( )`
- It tests the `clear( )` method of class `Inventory`.
- Every test class is derived from the class `TestCase`
- Each test method has to start with “test”
- One or more `Assert` statements are used to verify program behaviour.
- `Asserts` come in various flavours (see notes page)

# Running the test: TestInventory1

- Run the test using:

```
java junit.textui.TestRunner TestInventory1
```

Let us see the [output](#)

- If a test method contains multiple asserts, only the first failure is reported.

# Custom Asserts

- Custom asserts are possible by subclassing TestCase and using the [subclass](#) for all tests.
- Notice that ProjectTest is a customized subclass derived from class TestCase from which all test classes will be derived.
- This is a good practice because we can modify the subclass ProjectTest to add any functions that will be used by all the test methods.

# Failure, Error and Success

A failure is anticipated and checked for with assertions – e.g. we expect the inventory size to be 0, but it turns out to be 4.

Errors are unanticipated problems – e.g. the input data file is not found. These are reported by the JUnit framework.

We have a success if there is no failure or error – e.g. we expect the inventory size to be 0, but it does turn out to be 0.

The three test methods in [TestDemonstrateFailure.java](#) illustrate these.

Let us see the [output](#).

# More tests: TestInventory2

- [TestInventory2](#) illustrates a test class with multiple test methods:
  - testClear( ),
  - testAdd( ) and,
  - testSameManuf( )

- Run the test using:

```
java junit.textui.TestRunner TestInventory2
```

Let us see the [output](#). All tests pass.

# Test Fixtures

- Notice in TestInventory2, we read from a file to create an inventory at the start of a test and clear the inventory after the test.
- Such a common environment against which each test is run is called a test fixture.
- Test fixtures are automated by the setUp( ) and tearDown( ) methods.
- We see this in the file [TestInventory3.java](#)
- Let us see the [output](#) after running these tests.

# Test Suites: How to Include Methods from a Test Class

Test methods can be grouped into test suites.

- [TestSuite1.java](#) puts two tests from the class `TestInventory2` in a test suite.
  - `testClear`, and
  - `testAddItem`
- Study the JUnit statement(s) for doing this.
- Let us look at the [output](#).

# Test Suites: How to Include All Test Methods from a Test Class

- [TestSuite2.java](#) pulls all tests from the class `TestInventory2` into a test suite.
  - `testClear`
  - `testAddItem`, and
  - `testGetSameManuf`
- Study the JUnit statement(s) for doing this.
- Let us look at the [output](#).

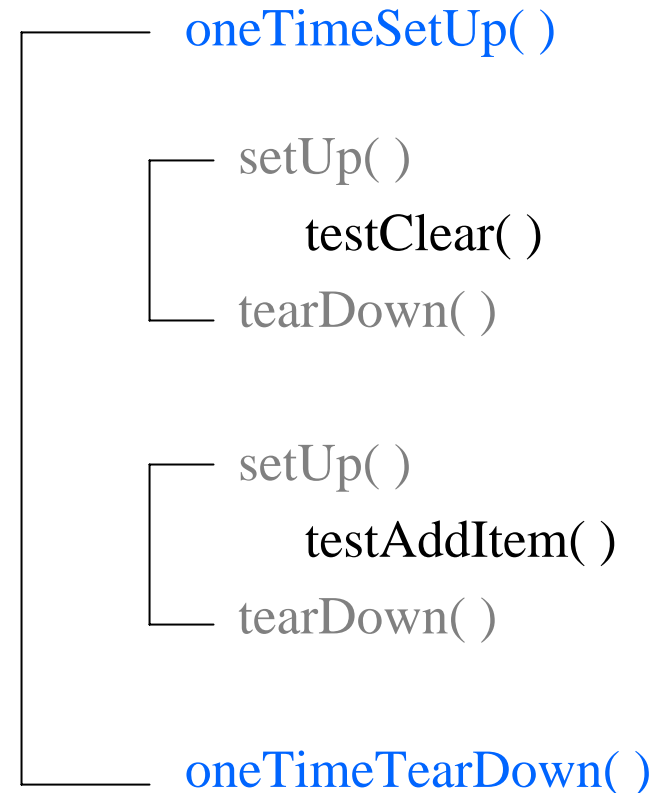
# Test Suites: How to Include Another Test Suite in a Test Suite

- Let us see how [TestSuite3.java](#) inserts one test suite (from the class TestSuite1) and one test method into another test suite.
- Here is the [output](#).

All these methods can be combined to create a composite test suite.

# One Time (per suite) Set Up Tasks

- One time set up is done before any test is run
- One time tear down is done after all tests have been run
- The set up and tear down methods associated with each test class are run before and after each test method from that class.



- [TestSuite4](#) illustrates one time set up and tear down methods
- Here is the [output](#).

# Design Lessons from JUnit

- **Design patterns**: A strategy to handle generic design issues which may appear in many software systems – for instance, ensuring that a class has only one instance and providing a global point of access to it – is a design issue that can come up in multiple development projects. This pattern is called the **singleton** pattern.
- There are many similar, more complex, design issues for which ready made solutions exist:
- **Reference**: “**Design Patterns: Elements of Reusable Object-Oriented Software**” by Erich Gamma, et al.
- The design of JUnit uses various design patterns and is an excellent example of object oriented design.
- **Reference**: **JUnit A Cook’s Tour** – available with the JUnit download.

# Mock Objects

- The unit of code being tested may depend on external systems.
- For instance, the unit may depend on system time.
- If a particular behaviour occurs only at 5 pm, then we would have to wait till 5 pm to test the item.
- For such situations, we may create mock objects to mimic the real system.
- Frameworks to create mock objects exist. Please see [www.mockobjects.com](http://www.mockobjects.com)
- Mock objects are not part part of the JUnit framework directly and will not be discussed further.

# Ant: Continuous Build and Test

- Ant is a Java based open source build facility available from [ant.apache.org](http://ant.apache.org).
- It provides tasks that support unit testing after each build
  - **Junit** task for running the unit tests and generating reports
  - **Junitreport** task to collect the consolidate all xml test results into one xml report (can be viewed with an internet browser)
  - **Mail** task to email the test report
- The junit and junitreport tasks are illustrated in the [build.xml](#) file. Let us go through this file.

# Ant: The Test Report

The `junitreport` task consolidates all xml test results into one [xml report](#) (can be viewed with an internet browser).

Six test methods have been run from two test classes.

There is one failure and one error – the reasons are given in the report.

# Test Quality: Code Coverage

- What is code coverage?

Determining those statements in the code which have been executed by a test run.

- Tells us if we have done enough testing.
- Does not address functional errors

- Part of the feedback loop in the development process

# Code Coverage with Clover

- Clover is a popular code coverage tool for Java programs, it can be integrated with Ant to provide continuous coverage information.
- 30 day evaluation copy downloadable from [www.cenqua.com](http://www.cenqua.com) - free to creators of open source software.

# Types of Code Coverage

Clover provides three types of coverage data:

1. Statement coverage – whether each statement is executed
2. Branch coverage – if a boolean expression in a control structure evaluated to both true and false
3. Method coverage – if each method was executed

# Clover with Ant

Clover can be integrated with Ant - [buildWithClover.xml](#)

Let us review the additions needed to use Clover.

The report tells us about the code coverage statistics of two classes:

- Item, and
- Inventory

Let us review the xml [coverage report \(index.html\)](#).

Additional tests are needed to get 100% coverage.

Review the coverage report and discuss what other tests are needed.

# Test Quality: Error Injection

Consider this change in the file inventory.java

```
...  
// return the number of items in the inventory  
public int size( )  
{  
    int counter = 1;  
    return items.size( );  
}
```

If the method `size( )` is invoked in a test, the test result will not be affected even if the value of `counter` is changed to 2. This is a defect – either `counter` is not needed, or another test is needed to check for the behavior of the `counter` variable.

# Testing JUnit Tests with Jester

- Before using Jester, ensure that all JUnit tests pass.
- Jester modifies the source, and check for test failure after each modification.
- Jester reports code changes that do not cause any test to fail.

Each change is undone before the next change is made.

- Always use a copy of the code while running Jester.
- Ignore the “false hits”, e.g. increasing the array bound will not fail a test. Jester reports it, but it is not an error.

# Testing JUnit Tests with Jester

Jester is created by Ivan Moore.

Download Jester from [jester.sourceforge.net](http://jester.sourceforge.net)

Jester modifies one Java source file at a time in one of the following ways:

- modifying literal numbers; e.g. 0 is changed to 1

- changing true to false and vice-versa

- changing if( to if(true ||                   // always true

- changing if( to if(false &&               // always false

# Running Jester with TestInventory3

We run Jester with the test class TestInventory3 and the source files Item.java and Inventory.java:

```
Java jester.TestTester TestInventory3 <source directory>
```

Let us look at the [results](#).

Although code coverage gives decent results, Jester shows large gaps in testing ☹️.

Review Jester output and see why our test cases are inadequate.

Suggest additional test cases.

# References

1. “Pragmatic Unit Testing in Java with JUnit”, A Hunt and D Thomas
2. “Unit Test Frameworks”, P Hamill
3. “JUnit Howto”, article by B Simpson
4. “JUnit A Cook’s Tour”, available with the JUnit download
5. “Test Infected”, available with the JUnit download
6. “Learning to Love Unit Testing”, article by A Hunt and D Thomas
7. “Jester – a JUnit test tester”, article by Ivan Moore