



JUnit Framework

© FPT Software

1

Latest updated by: HanhTT1

Introduction

- ↳ **Duration:** 2 hours
- ↳ **Purpose:** introduce about unit testing and JUnit
- ↳ **Audience:** developers
- ↳ **Test:** Practice test

Objectives

After the course, student will:

- ↳ Understand Unit testing and Junit
- ↳ Know how to write a JUnit test class
- ↳ Know how to install JUnit and run a JUnit test case
- ↳ Know about some tips of Unit testing



Agenda

- Introduction
- What is JUnit?
- Setting up JUnit
- Exploring JUnit
- Samples
- Calculate Code coverage
- Tips

Notice: This training is for JUnit 3

Introduction

* You know

Every programmer knows they should write tests for their code.

* But

The universal response to "Why not?" is "I'm in too much of a hurry."

JUnit comes as a solution for this situation

What is JUnit?

- JUnit is de facto Java unit testing framework
- Integrated nicely with many IDEs, Ant
- Easy to use and to learn

Agenda

- Introduction
- What is JUnit?
- **Setting up JUnit**
- Exploring JUnit
- Samples
- Calculate Code coverage
- Tips

Setting up JUnit

★ Download JUnit from <http://junit.org/>

★ Installation

- Unzip the junit.zip file
- Add **junit.jar** to the CLASSPATH.

★ Testing

Test the installation by using either the batch or the graphical TestRunner tool to run the tests that come with this release.

All the tests should pass OK.

Notice:

- The tests are not contained in the junit.jar but in the installation directory directly. Therefore make sure that the installation directory is on the class path

Setting up JUnit (cont)

★ Testing

- for the console TestRunner type:

java junit.textui.TestRunner junit.samples.AllTests

- for the graphical TestRunner type:

java junit.awtui.TestRunner junit.samples.AllTests

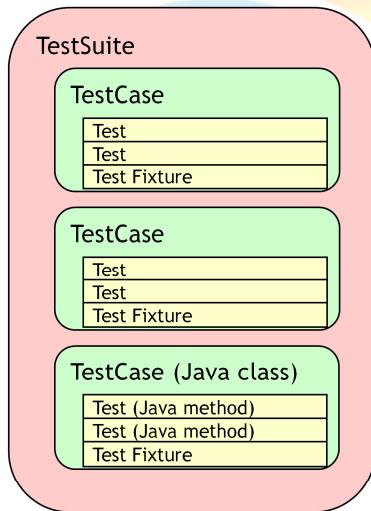
- for the Swing based graphical TestRunner type:

java junit.swingui.TestRunner junit.samples.AllTests

Agenda

- Introduction
- What is JUnit?
- Setting up JUnit
- **Exploring JUnit**
- Samples
- Calculate Code coverage
- Tips

Exploring JUnit



- Test runner → Result
- When you want to write JUnit test for a class, you need to implement a **TestCase** object
 - A **test (Java method)** tests a single method of the class
 - * You can have multiple test for a single method
 - The **test fixture** provides the set of common resources or data that you need to run one or more tests.
 - When you need to run several TestCase objects at once, you create another object called a **TestSuite**
 - The **test runner** runs tests or an entire test suite

© FPT Software

11

- When you write JUnit test, most often you write TestCases and Tests

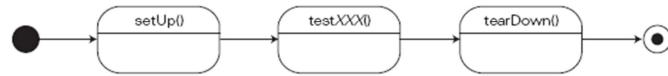
Implementing a Test Case class

- ★ Extend `junit.framework.TestCase` class
- ★ Inherit the following methods:
 - `protected void setUp()`
 - Typically, you will override this method and use it to initialize the test fixture you need in testing
 - `protected void tearDown()`
 - You will override this method to release or destroy test fixtures if need
- ★ Write any number of test (test methods), all of which have the form `public void testXXX()`
- ★ Define a `main()` method that runs the `TestCase` class
- ★ A typical `TestCase` includes two major components:
 - the fixture
 - the tests

- The framework ships with ready-to-use graphical and textual TestRunners. The framework can also generate a default runtime TestSuite for you. So, the only class you absolutely must provide yourself is the `TestCase`.

Test Fixture

- ★ The set of background resources that you need to run a test is commonly called a *test fixture*. Eg: database connection, file, test data...
- ★ A fixture is automatically created and destroyed by a TestCase through its setUp and tearDown methods.
- ★ Order when run a test method



- A database connection is a good example of why you might need a fixture. If a TestCase includes several database tests, they each need a fresh connection to the database. A fixture makes it easy for you to open a new connection for each test without replicating code.

-

Creating a Test

- ★ Your test method should start with *public void testSomething()*, where **Something** is any name you like (typically the name of the method you are testing)
- ★ Fundamental steps to write a Test:
 - Initiate necessary fixtures
 - Call the method being tested and get the actual result
 - Assert what the correct result should be with one of the **assert methods** of JUnit frame work
 - These steps can be repeated as many times as necessary

- The framework ships with ready-to-use graphical and textual TestRunners. The framework can also generate a default runtime TestSuite for you. So, the only class you absolutely must provide yourself is the TestCase. A typical TestCase includes two major components: the fixture and the unit tests.
- You do not need to call these methods, they will be called automatically
- You do not need to do anything if the tests fail; the framework will take care of this for you

Assert methods

- ★ An assert method is a JUnit method that performs a test, and throws an AssertionFailedError if the test fails.
- ★ There are 8 core assert methods:
 - `assertTrue`: Asserts that a condition is true
 - `assertFalse`: Asserts that a condition is false.
 - `assertEquals`: Asserts that two objects are equal.
 - `assertNotNull`: Asserts that an object isn't null.
 - `assertNull`: Asserts that an object is null.
 - `assertSame`: Asserts that two objects refer to the same object.
 - `assertNotSame`: Asserts that two objects do not refer to the same object.
 - `fail`: Fails a test immediately with the given message.

The Counter class

```
public class Counter {  
    int count = 0;  
  
    public int increment() {  
        return ++count;  
    }  
  
    public int decrement() {  
        return --count;  
    }  
  
    public int getCount() {  
        return count;  
    }  
}
```

- The constructor will create a counter and set it to zero
- The increment method will add one to the counter and return the new value
- The decrement method will subtract one from the counter and return the new value

JUnit test for Counter

```
public class CounterTest extends junit.framework.TestCase {  
    Counter counter;  
  
    public CounterTest() { } // default constructor  
  
    protected void setUp() { // creates a (simple) test fixture  
        counter = new Counter();  
    }  
  
    protected void tearDown() { } // no resources to release  
  
    public void testIncrement() {  
        assertTrue(counter.increment() == 1);  
        assertTrue(counter.increment() == 2);  
    }  
  
    public void testDecrement() {  
        assertEquals(counter.decrement(), -1);  
    }  
  
    public static void main(String[] args) {  
        junit.textui.TestRunner.run(CounterTest .class);  
    }  
}
```

Note that each test begins with a *brand new* counter

This means you don't have to worry about the order in which the tests are run

Creating a TestSuite

- ★ Create a class and implement suite() method to create TestSuite and add TestCase classes to TestSuite

```
public static Test suite() {  
    TestSuite suite = new TestSuite("test");  
    suite.addTest(Counter.class);  
    suite.addTest(Caculator.class);  
    return suite;  
}
```

- ★ Define a main() method that runs the TestSuite

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

Test Runner

- ★ The JUnit distribution includes three TestRunner classes:

- For the text console: *junit.textui.TestRunner*
- For Swing: *junit.swingui.TestRunner*
- for AWT: *junit.awtui.TestRunner*

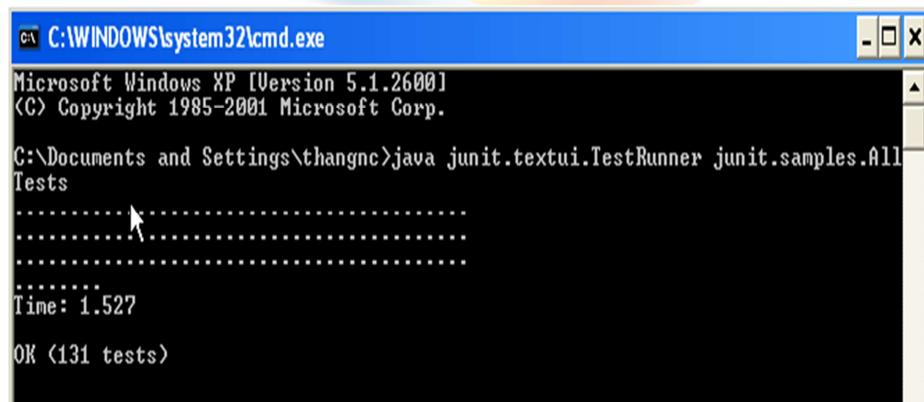
- ★ Run a TestSuite or a TestCase

- java *junit.textui.TestRunner* suite class/testcase class
- java *junit.swingui.TestRunner* suiteclass/testcase class
- Java *junit.awtui.TestRunner* suiteclass/testcase class

- ★ Many IDEs support their own TestRunner to run:
Eclipse, JDeveloper...

Test Runner (cont)

★ Text console



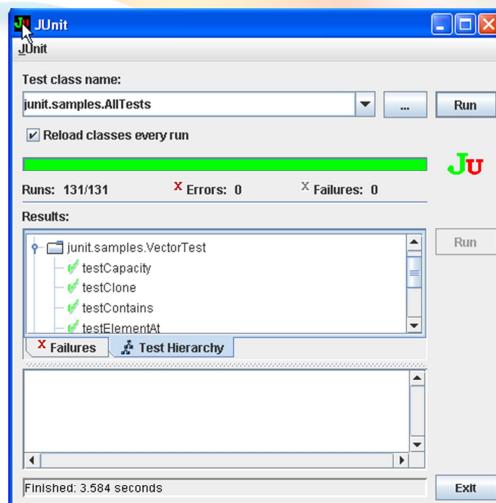
A screenshot of a Microsoft Windows XP command prompt window titled "cmd C:\WINDOWS\system32\cmd.exe". The window shows the output of a JUnit test run. The text in the window reads:

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\thangnc>java junit.textui.TestRunner junit.samples.AllTests
.....
Time: 1.527
OK (131 tests)
```

Test Runner (cont)

★ Swing

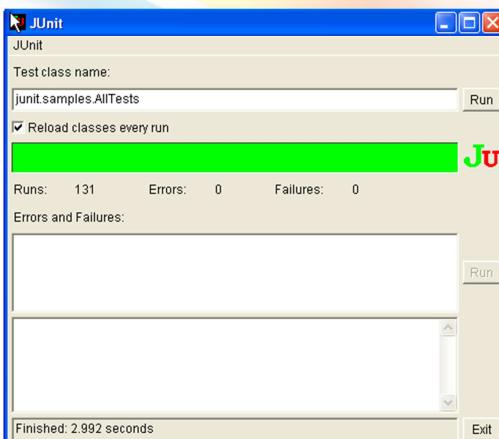


© FPT Software

21

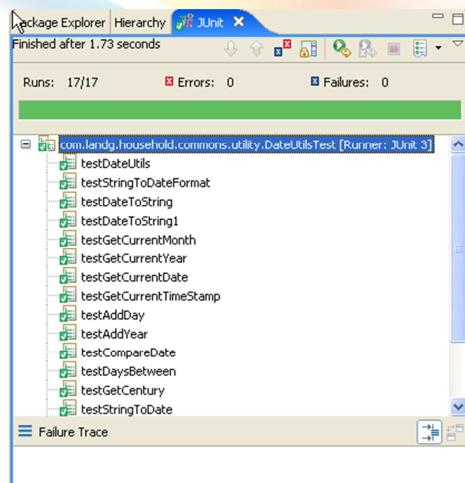
Test Runner (cont)

★ Awt



Test Runner (cont)

★ Eclipse



© FPT Software

23

Test Result

★ A test method in a test case can have one of three results:

- Pass – all assertions matched expected values
- Failed – an assertion did not match expected value
- Error – an unexpected exception was thrown during execution of test method

Agenda

- Introduction
- What is JUnit?
- Setting up JUnit
- Exploring JUnit
- **Samples**
- Calculate Code coverage
- Tips

Sample

* **Demo Shopping Cart sample**



Demo

Agenda

- Introduction
- What is JUnit?
- Setting up JUnit
- Exploring JUnit
- Samples
- Calculate Code Coverage
- Tips

Code coverage

- ★ Good unit tests are thorough; they test everything that's likely to break. You can aim to test every line of code, every possible branch the code might take, every exception it throws, and so on
- ★ Use code coverage tools to determine how much of the code under test is actually being exercised (Clover, Cobertura, EMMA ...)

Cobertura tool

The screenshot shows the Cobertura tool interface displaying a coverage report for the `Product` class. The left sidebar lists packages and classes, with `Product` highlighted at 90% coverage. The main pane shows the code with colored highlights indicating coverage status: green for covered lines, red for uncovered lines, and grey for lines not analyzed. The report includes a summary table and the generated Java code.

Coverage Report - com.fsoft.training.junit.Product

Classes in this File	Line Coverage	Branch Coverage	Complexity
Product	90%	50%	1.5

```
1 package com.fsoft.training.junit;
2
3 public class Product {
4     private String title;
5     private double price;
6
7     public Product(String title, double price) {
8         this.title = title;
9         this.price = price;
10    }
11
12    public String getTitle() {
13        return title;
14    }
15
16    public double getPrice() {
17        return price;
18    }
19
20    public boolean equals(Object o) {
21        if (o instanceof Product) {
22            Product p = (Product)o;
23            return p.getTitle().equals(title);
24        }
25
26        return false;
27    }
28
29 }
```

Report generated by Cobertura 1.9 on 11/14/07 4:28 PM.

© FPT Software

29

Cobertura tool

★ What is **Cobertura**?

- free Java tool that calculates the percentage of code accessed by tests.
- used to identify which parts of your Java program are lacking test coverage.

★ Features

- Can be executed from ant or from the command line.
- Instruments Java bytecode after it has been compiled.
- Can generate reports in HTML or XML.
- Shows the percentage of lines and branches covered for each class, each package, and for the overall project.
- Can sort HTML results in ascending or descending order by
 - class name
 - percent of lines covered
 - percent of branches covered

★ Download at : <http://cobertura.sourceforge.net/>

Cobertura tool

★ How to setup ?

- Setting value for system values
 - JAVA_HOME
 - ANT_HOME
 - Include ANT_HOME/bin and JAVA_HOME/bin to PATH variable
 - Point CLASSPATH to necessary Jar, zip files
- Create build properties file – Configure report structure
- Create Ant build file – build.xml file

★ Demo **Shopping Cart sample**



Demo

Agenda

- Introduction
- What is JUnit?
- Setting up JUnit
- Exploring JUnit
- Samples
- Calculate Code coverage
- **Tips**

Tips (Best practices)

- ★ Do not use the test-case constructor to set up a test case
- ★ Don't assume the order in which tests within a test case run
- ★ Call a superclass's setUp() and tearDown() methods when subclassing
- ★ Utilize JUnit's assert/fail methods and exception handling for clean test code
- ★ Keep tests small and fast
- ★ Do not load data from hard-coded locations on a filesystem
- ★ Name tests properly
- ★ For each Java package in your application, define a TestSuite class that contains all the tests for validating the code in the package.
- ★ Define similar TestSuite classes that create higher-level and lower-level test suites in the other packages (and sub-packages) of the application.
- ★ Make sure your build process includes the compilation of all tests.

Mock Object

- ★ Unit-testing each method in isolation from the other methods or the environment is certainly a nice goal. But how do you perform this feat?
- ★ The answer is called *mock objects*.
- ★ A *mock object* (or *mock* for short) is an object created to stand in for an object that your code will be collaborating with. Your code can call methods on the mock object, which will deliver results as set up by your tests.
- ★ Some mock object frame works:
 - EasyMock: www.easymock.org/
 - jMock: www.jmock.org
 - Mocquer: mocquer.dev.java.net/

JUnit extensions

- ★ There are many JUnit extensions supporting for implementing JUnit test:
 - DBUnit to test database
 - HttpUnit, JWebUnit, Canoo WebTest to test interface
 - XMLUnit to test XML
 - StrutsTestCase to test Struts
 - ...

CONCLUSION

- JUnit is de facto Java unit testing framework
- Use TestRunner, Ant or IDEs to run test cases
- Extend junit.framework.TestCase class
- setUp() method is used to initialize test case
- tearDown() is used to release or destroy
- Use TestRunner or IDEs to run your test cases
- There are many JUnit extensions such as DBUnit, HttpUnit, StrutsTestCase ...

Resources & references

- Resources
 - www.junit.org/
 - <http://junit.sourceforge.net>
- Recommended readings
 - Manning – JUnit in Action
 - Test Driven Development: By Example. Boston: Addison-Wesley, 2003



© FPT Software

38