

JDBC & Tools

Coding Standard and JUnit

Module 3

Agenda

1

File Organization

2

Naming Conventions

3

Documentation

4

Formatting

5

Java Beans

Agenda (Contd.).

- 6 Introduction to JUnit**
- 7 JUnit with Eclipse**
- 8 Assert methods and Annotations**
- 9 Parameterized test**
- 10 Test Suite**

Objectives

In this module, you will be able to:

- Implement Naming Conventions for Java
- Implement Documentation Standards
- Implement Formatting Standards
- Implement Java Bean Conventions
- Explore JUnit?
- Learn about how to write Test cases?
- Implement the various assert methods
- Carry on Parameterized tests
- Understand Test Suites

File Organization

Java Source Files

- File Header
- Package Declaration
- Import Declarations
- Class or Interface Declarations

File Header

- Every java source file should have a file header and the file header should be the first section. File header is meant for capturing the file specific information like copyright, file name, short description, version number, created date, and modification history. File header information is meant to be automatically created by the source code control system. This is another reason why the file header should be the top section in the source file. **(Rule Category: Mandatory)**

```
/*  
 * <Copyright information>  
 *  
 * <Customer specific copyright notice (if any) >  
 *  
 * <File Name>  
 *  
 * <Short Description>  
 *  
 * <Version Number >  
 *  
 * <Created Date in dd mmm yyyy format>  
 *  
 * <Modification History>  
 */
```

Package Declaration

- The first non-comment line of a Java source file should be the package declaration. Package statement is mandatory for every Java source file. No Java source file should belong to the "default" package. Example of a package statement is below. **(Rule Category: Mandatory)**

```
package com.wipro.generic;
```

- Usage of default modifier (package specific) is highly discouraged. Explicit specification of modifier signifies intent to the reader of your code. It also makes the code more modularized, secure and maintainable. As a good practice make the code as inaccessible as possible

```
public class Test{ //Fixed
    void method1(){
        System.out.println("Inside method");
    }
}
```


Import Declaration

- Import declarations follow the package declaration. Import declarations should be grouped together by the package name. For unqualified access of the static members of the class, *static import should follow immediately after the import of all the classes, a line should be added to separate it from class imports. Within a group the import statements should be sorted lexically. A single blank line should be added to separate the groups if required.

```
import java.util.Date;
import java.util.Timer ;
import org.w3c.dom.Document ;

import com.wipro.generic.CodeEncoder;
import com.wipro.generic.TextConverter;

import static java.util.Calendar.AM;
import static java.util.Calendar.AM_PM;
```

Classes or Interface Declarations

- Class or Interface Declarations follow the import declarations section. A java source file can contain one or more class/interface declarations (a.k.a. type declarations). However, there should be at the maximum one public declaration per file. Also the public class/interface declaration should be the first declaration in a source file. Every type declaration is preceded by a documentation comment. **(Rule Category: Mandatory)**

```
import java.util.Date;  
import java.util.Timer ;  
  
public class SomeClass{}  
class SomeOtherClass{}
```

Class / Interface Organization

- Class or Interface documentation comment (`/**...*/`)
 - This is a javadoc style comment and will start on the same column as the Class or Interface statement that follows it. **(Rule Category: Mandatory)**
- Class (static) variables (member fields)
 - First the **public** class variables, then the **protected**, then the default (no access modifier), and then the **private** static variables should be declared. **(Rule Category: Optional)** - Each static variable should be declared in a line by itself. **(Rule Category: Mandatory)**
- Instance variables (member fields)
 - First **public**, then **protected**, then default (no access modifier), and then **private** instance variables should be declared. **(Rule Category: Optional)**. Each instance variable should be declared in a line by itself. **(Rule Category: Mandatory)**
- Static Member Inner class declarations
 - Each static member inner class should have a class header (javadoc comment for the class) immediately preceding it. **(Rule Category: Mandatory)**

Class / Interface Organization (Contd.).

- Instance Constructors
 - Each constructor should have a method header (javadoc comment for the method) immediately preceding it. **(Rule Category: Mandatory)**
- Instance Member Inner class declarations
 - Each instance member inner class should have a class header (javadoc comment for the class) immediately preceding it. **(Rule Category: Mandatory)**
- Instance Methods
 - The methods in a class or interface should be **grouped by functionality** rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier. **(Rule Category: Optional)**
 - Each method should be preceded by a javadoc comment explaining description of the method, return variables and parameters. **(Rule Category: Mandatory)**

Quiz

1. The first non-comment line of a java source file is

- a. Package declaration
- b. File header
- c. import statement
- d. class and interface declaration

Package declaration

2. Which of the following is the class or interface documentation comment?

- a. //
- b. /* */
- c. /** .. */
- d. None of the above

/** .. */

Naming Conventions

Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the purpose of the identifier--for example, whether it is a constant, package, or class--which can be helpful in understanding the code

- Packages

- Package names should be single lowercase words. Package names should not contain words containing capital letters. From the package name the keyword 'package 'should be separated by a Single Space and first column should start with package statement. For packages that will be widely distributed, unique package prefix scheme should be used. Prefix the site DNS name in reverse order to the package to make it unique. For example, 'com.wipro' should be used for naming packages in Wipro. **(Rule Category: Mandatory)**

- Source Files

- Source Files name should be same as Class Name (See Classes row for details). Each source file should contain only one public named class to avoid difficulties in figuring out the source file while coding and debugging. **(Rule Category: Mandatory)**

Naming Conventions (Contd.).

- Classes and Interfaces
 - A class name should be in mixed case with the first letter capitalized and the first letter of each internal word capitalized. Class names should be nouns or noun phrases.
 - Try to keep your class names simple and descriptive. Use mnemonic whole words--avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).
 - SessionBean or MessageBean should be suffixed by Bean. The Local Interface of a Session EJB should be suffixed by 'Local'. The Local Home interface of a Session EJB should be suffixed by 'LocalHome'. Remote Interface of a Session EJB should NOT be suffixed. **(Rule Category: Mandatory)**
- Annotations
 - Same as class or interface naming conventions **(Rule Category: Mandatory)**

Naming Conventions (Contd.).

- Methods

- A method name should be in mixed case with the first letter lowercase and the first letter of each internal word capitalized. **(Rule Category:**

Mandatory)Avoid using short method names**(Rule Category: Mandatory)**

- Classes should not have non-constructor methods with the same name as the class**(Rule Category: Mandatory)**

- Getter is a method that returns the value of an attribute. Prefix the word 'get' to the name of the attribute to form the name of a getter. Setter is a method that modifies the value of an attribute. Prefix the word 'set' to the name of the attribute to form the name of a setter. **(Rule Category: Mandatory)**

- Methods which are expected to return a Boolean should be formed as a question which can be answered with a Boolean. **(Rule Category: Mandatory)**

Naming Conventions (Contd.).

- Variables
 - Naming conventions are the same for instance variables, class (static) variables, variables local to a method and method parameters. They all should be in mixed case with a lowercase first letter. Internal words start with capital letters. **(Rule Category: Mandatory)**
 - Integer literals should not start with zero. Zero means that the rest of literal will be interpreted as an octal value. **(Rule Category: Mandatory)**
 - Avoid excessively long variable names **(Rule Category: Mandatory)**
- Final Static Variables
 - Constants, values that do not change, should be implemented as **static final** attributes of classes. **(Rule Category: Mandatory)**
 - Use all upper case letters with internal words separate by underscores. **(Rule Category: Mandatory)**

Naming Convention (Contd.).

- **Exception Objects**

- Because exception handling is very common in Java, the letter 'e' should be used for a generic exception. However, when using multiple exception types in a single try catch block use an abbreviated name of the Exception in small case followed by 'e' **(Rule Category: Mandatory)**

Quiz

Which of the following are true regarding Naming Conventions?

- a. The Package names should be in Capital letters
- b. The Package names should be in Lowercase letters
- c. Classes and interfaces should have mixed case (first letter Capital)
- d. Class names should be verbs
- e. Final variables should be all in lowercase

b and c are correct

Documentation

Documentation

- Java language supports two types of comments: documentation and implementation comments.
- Documentation comments are delimited by `/** ... */`. Javadoc comments are meant to describe the specification of code, from an implementation free prospective to read by developers.
- The other type of comments includes two other styles, which are delimited by `/* ... */` and `//`.
- These types of comments are used either for explaining the developers thinking behind a particular piece of code or to comment out unused code.

Documentation Comments

- Always first line comment should be in simple characters `/**` without any text on the line and should be aligned with the following declared entity.
- Remaining lines may consists of an asterisk, single space, comment text and aligned with the first asterisk of the first line.
- Comment text in the beginning sentence is special and should be a self contained summary sentence. (Rule Category: Mandatory)
- Class and interface comments can use the `@version`, `@author`, and `@see` tags, in that order. If there are multiple authors, use a separate `@author` tag for each one. Required tags: `@version`, `@author` tags. (Rule Category: Mandatory)
- Constructor comments can use the `@param`, `@exception`, and `@see` tags, in that order. Required tags: one `@param` tag for each parameter and one `@exception` tag for each exception thrown. (Rule Category: Mandatory)

Documentation Comments (Contd.).

- Method comments can use the @param, @return, @exception, and @see tags, in that order. Required tags: one @param tag for each parameter, one @return tag if the return type is not void, and one @exception tag for each exception thrown. (Rule Category: Mandatory)
- Variable comments can use any javadoc tags, which are applicable, should be used. These javadoc tags are @values, @concurrency, @see, @example, @fyi, etc. Required tags: none. (Rule Category: Mandatory)

Comments for Class or Interface

```
/**
 * A facility for threads to schedule tasks for future execution in a
background thread. Tasks may
 * be scheduled for one-time execution, or for repeated execution at
regular intervals.
 *
 * <p>If the timer's task execution thread terminates unexpectedly, for
example, because its
 * <tt>stop</tt> method is invoked, any further attempt to schedule a task
on the timer will
 * result in an <tt>IllegalStateException</tt>, as if the timer's
<tt>cancel</tt> method had
 * been invoked.
 *
 * <p>This class is thread-safe: multiple threads can share a single
<tt>Timer</tt> object without
 * the need for external synchronization.
 *
 * <p>Implementation note: This class scales to large numbers of
concurrently scheduled tasks
 * (thousands should present no problem). Internally, it uses a binary
heap to represent its
 * task queue, so the cost to schedule a task is  $O(\log n)$ , where  $n$  is the
number of concurrently
 * scheduled tasks.
 *
 * @author Josh Bloch
 * @version 1.9, 01/23/03
 * @see TimerTask
 * @see Object#wait(long)
 * @since 1.3
 */
public class Timer {
```

Comments for method

```
/**
 * Checks a object for "coolness". Performs a comprehensive
 * coolness analysis on the object. An object is cool if it
 * inherited coolness from its parent; however, an object
can
 * also establish coolness in its own right.
 *
 * @param obj the object to check for coolness
 * @param name the name of the object
 * @return true if the object is cool; false otherwise.
 * @exception OutOfMemoryError If there is not enough
memory to
 * determine coolness.
 * @exception SecurityException If the security manager
cannot be
 * created
 * @see isUncool
 * @see isHip
 */
public boolean isCool(Object obj, String name)
    throws OutOfMemoryError, SecurityException {
```

Block Comments

- A regular block comment starts with the characters `/*` and ends with the characters `*/`. A block comment should be preceded by a blank line to set it apart from the rest of the code. **(Rule Category: Mandatory)** Block comments have an asterisk `"*` at the beginning of each line except the first.

```
/*  
 * Look for preexisting entry for key. This will never happen for  
 * clone  
 * or deserialize. It will only happen for construction if the  
 * input Map  
 * is a sorted map whose ordering is inconsistent w/ equals.  
 */  
for (Entry e = table[i]; e != null; e = e.next) {  
    if (e.hash == hash && eq(k, e.key)) {  
        e.value = value;  
        return;  
    }  
}
```

Java Documentation

- Javadoc Method

- Java documentation should not contain syntax errors. (**Rule Category: Mandatory**)

Use the correct syntax for the @throws tag:

```
/** @throws exception_class_name text */
    public class SampleFixed {
        /** @throws IOException */    // FIXED
        FileInputStream openFile (String path) throws
IOException{
            System.in.read ();
            return null;
        }
        /** @throws $none */    // FIXED
        static void method () {
        }
    }
```

@ return tag in Javadoc Method

- Every method which returns a data type should contain @return statement in the java documentation. (**Rule Category:Mandatory**)

```
public class Sample {  
    /**  
     * This method print the message and return the integer  
     * value.          * @return int - of value.          // FIXED  
     */  
    public int SampleMethod() {  
        int a = 10;  
        //To-Do (implementattion for "Sample");  
        return a*a;  
    }  
}
```

@ param tag in Javadoc Method (Contd.).

- In any public method written in the java code which contains parameters, the java documentation should contain @param tag for specifying the parameters used in the method. (**Rule Category : Mandatory**)

```
public class ParamTest {  
    /**  
     * This method print the message and return the  
     integer value.      * @param b - Takes the int  
     value as parameter      // FIXED      */  
    public void SampleMethod(int b) {  
        int a = b;  
        //To-Do (implementattion for "Param Value :" +  
a);  
    }  
}
```

Formatting

Indentation

- One of the ways to improve the code readability is to group the individual statements into block statements and uniformly indent the content of each block to set off its contents from the surrounding code. Nested code should be indented one further level than the parent block. **(Rule Category: Mandatory)**
- Four spaces should be used as the unit of indentation. Most of the IDE tools allow the interpretation of tab for indentation. In such cases set the tab size to 4. If the IDE doesn't support tab to white space interpretation do not use the tab character for indentation. Four spaces are generally accepted as the indentation size. However this can be configured on a project basis if there is a strong reason to do so. **(Rule Category: Mandatory)**

Wrapping Lines

- The length of a line should not be more than 80 characters. 80 characters of line size is generally accepted. However, it can be configured on a project basis if there is strong reason to do so. **(Rule Category: Mandatory)**
 - Break after a comma
 - Break before an operator
 - Align the new line with the beginning of the expression at the same level on the previous line
 - If the above rules lead to confusing code or to code that is squished up against the right margin, just indent 8 spaces instead

```
methodCall(longExpression1, longExpression2, longExpression3,  
           longExpression4, longExpression5); //aligned with  
the new line  
var = method1(longExpression1,  
              method2(longExpression2,  
                      longexpression3));    //indented with  
the new line
```

White Space Usage

- Use single space immediately after right parenthesis ')' or a right curly brace '}' to separate any keyword that follows. Similarly a single white space should be used before the left parenthesis '(' or a left curly brace '{' when follow a keyword. **(Rule Category: Mandatory)**
- Using single space with any binary operator, except for the "." qualification operator and the expression that proceeds and the expression that follows the operator. **(Rule Category: Mandatory)**
Please see the example below.
- Use blank lines to separate:
 - Each logical section of a method implementation
 - Section / member of a class and / or an interface definition

```
double diagonal = Math.sqrt(x * x + y * y);  
double norm = length > 0.0 ? (x / length) : x;
```

Curly Braces

- Brace should be placed at the end of the statement on the same line (Kernighan & Ritchie style) (used as standard in this document). The closing brace should be on a line by itself aligned with the indentation level of the original statement or declaration. **(Rule Category: Mandatory)**

Java Beans

Java Beans Conventions

- Java Bean class must be a member of a package
- The Bean class should be declared as a public class
- Instance variables of the bean class must be private
- The bean class should provide some public access methods (get / set methods) to access the private instance variables.
- Bean class must have a public zero argument constructor.

Quiz

Why should we follow Coding Conventions?

80% of the lifetime of a software goes in maintenance

Hardly any software is maintained for its whole life by the original author

Code conventions improve the readability of the software, allows engineers to understand the new code more quickly and thoroughly

Which of the following are correct way of declaration?

a. `class eg{
 }`

b. `interface eg{
 }`

c. `class WiproEmployee{
 }`

```
class WiproEmployee{  
}
```

Introduction to JUnit

Introduction

- JUnit is an open source testing framework for Java
- It is a simple framework for creating automated unit tests
- JUnit test cases are Java classes that contain one or more unit test methods
- These tests are grouped into test suites
- JUnit tests are pass/fail tests explicitly designed to run without human intervention
- JUnit can be integrated with several IDEs, including Eclipse
- The JUnit distribution can be downloaded as a single jar file from <http://www.junit.org>
- It has to be kept in the classpath of the application to be tested

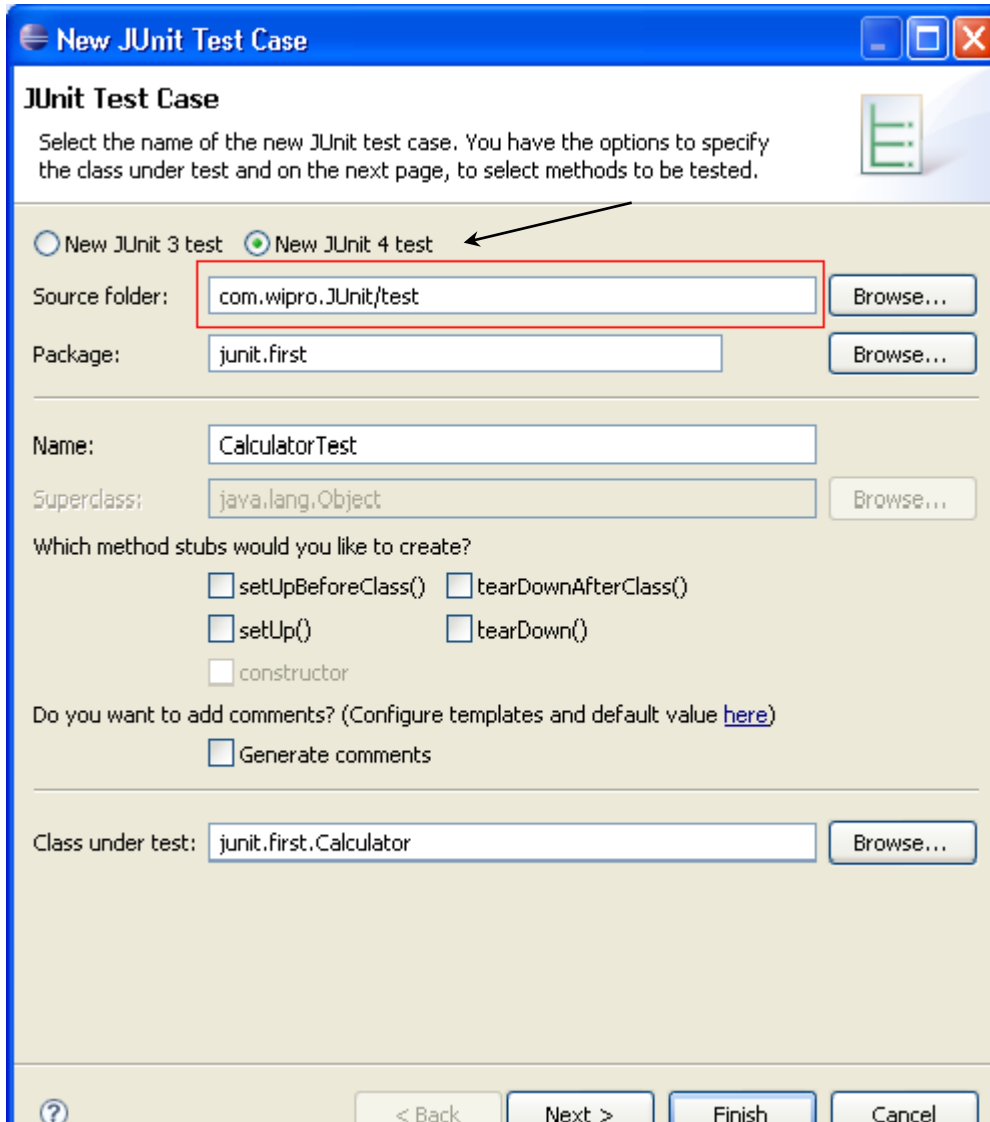
JUnit with Eclipse

JUnit with Eclipse

- Create a new Project com.wipro.JUnit
- Add JUnit.jar to the Classpath
- Right click the Project and create a new Source folder called 'test'
- Create a new Java class called Calculator in a package junit.first
- Add 2 methods add and sub to the Calculator class which does addition and subtraction of 2 numbers respectively

```
package junit.first;  
public class Calculator {  
    public int add(int x,int y)  
    { return x+y; }  
    public int sub(int x,int y)  
    { return x-y; } }
```

JUnit with Eclipse (Contd.).



New JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

☐ New JUnit 3 test ☒ New JUnit 4 test

Source folder: Browse...

Package: Browse...

Name:

Superclass: Browse...

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()
☐ setUp() ☐ tearDown()
☐ constructor

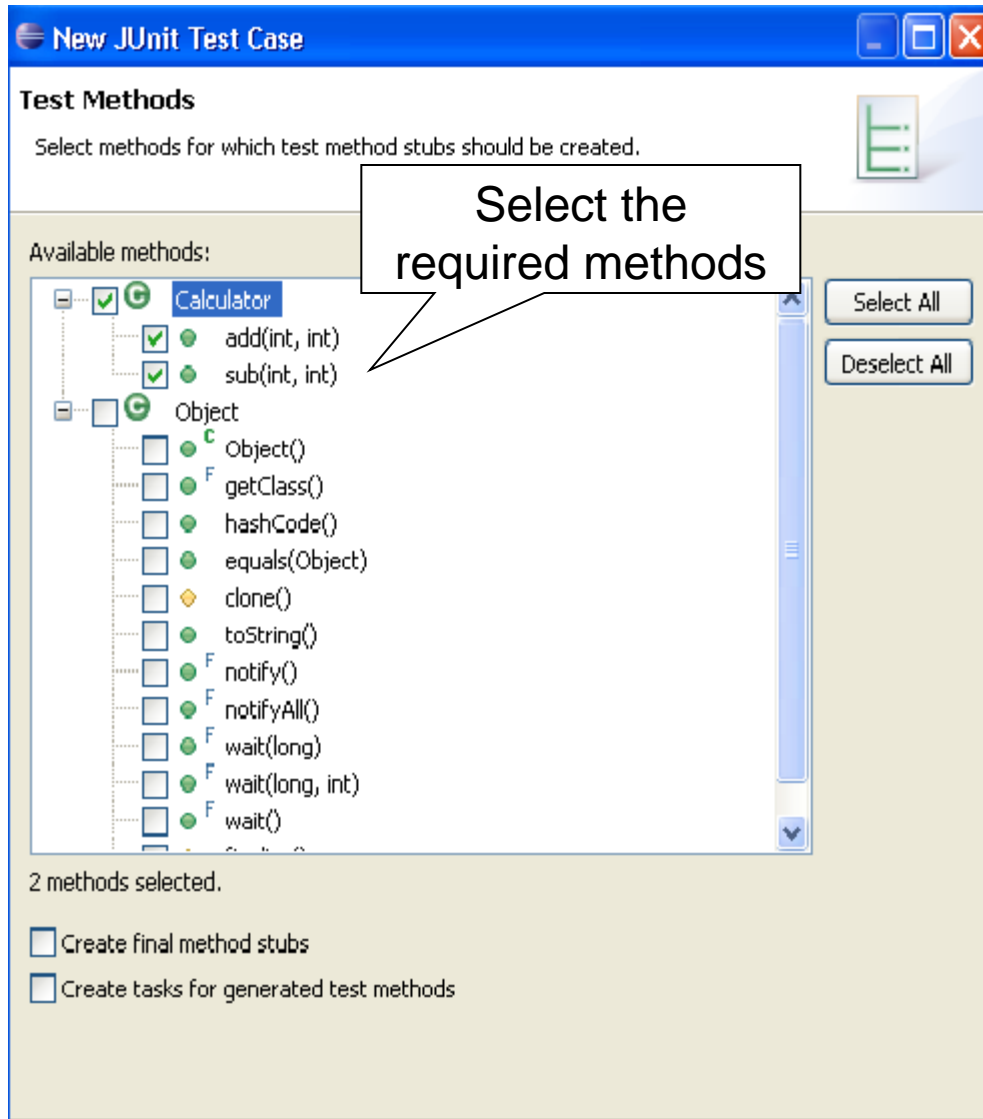
Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

Class under test: Browse...

< Back Next > Finish Cancel

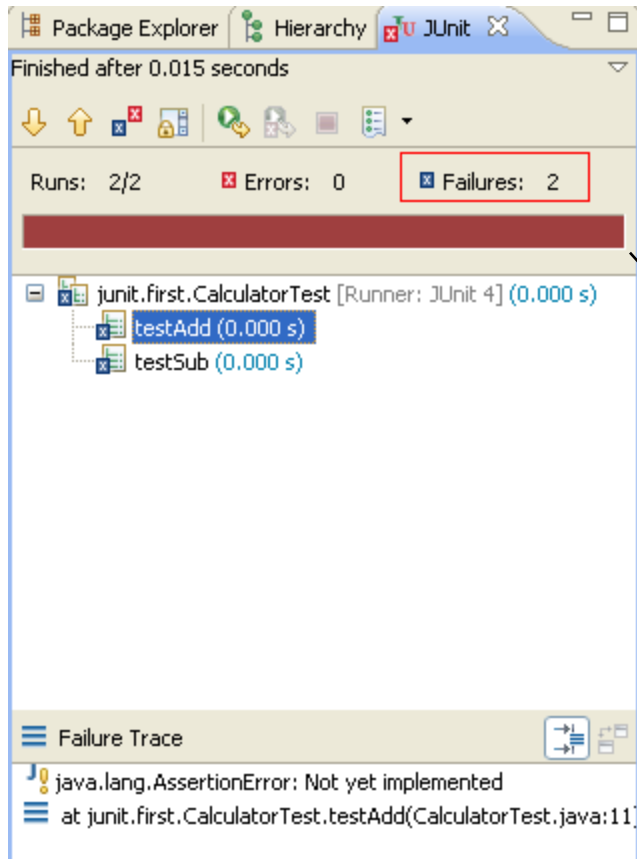
- Right click on the Calculator class in the Package Explorer and select New->JUnitTestCase
- select "New JUnit4 test"
- set the source folder to "test" – the test class gets created here

JUnit with Eclipse (Contd.).



- Press "Next" and select the methods you want to test

JUnit with Eclipse (Contd.).



- Right click on CalculatorTest class and select
Run-As → JUnit Test
- The results of the test will be displayed in JUnit view
- This is because the testAdd and testSub are not implemented correctly

Brown color indicates failure

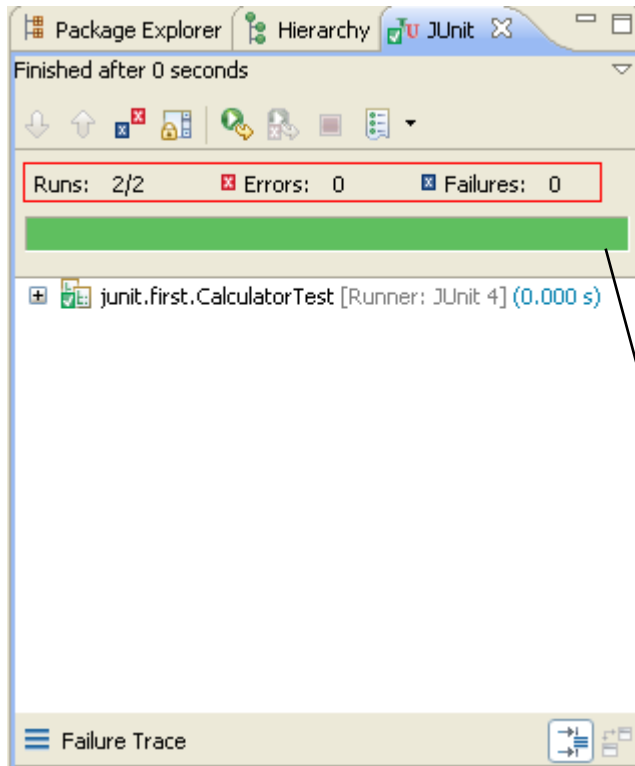
How to write a JUnit test method

- All the test methods should be marked with the JUnit annotation - `@org.junit.Test`
- All the JUnit test methods should be "public" methods
- The return type of the JUnit test method must be "void"
- The test method need not start with the test keyword
- Here is a simple JUnit test method:

```
@Test
public void testAdd()
{
    Calculator c=new Calculator();
    assertEquals("Result",5,c.add(2,3));
}
```

JUnit with Eclipse

- Now let's provide implementation to the code and run the test again



```
package junit.first;
import static org.junit.Assert.*;
import org.junit.Test;
public class CalculatorTest {
    @Test
    public void testAdd() {
        Calculator c=new Calculator();
        assertEquals(5,c.add(2,3));
    }
    @Test
    public void testSub() {
        Calculator c=new Calculator();
        assertEquals(20,c.sub(100,80));
    }
}
```

Green color indicates pass

Assert methods and Annotations

Assert methods with JUnit

- **assertArrayEquals()**

- Used to test if two arrays are equal to each other

```
int[] expectedArray = {100,200,300};  
int[] resultArray = myClass.getTheIntArray();  
assertArrayEquals(expectedArray, resultArray);
```

- **assertEquals()**

- It compares two objects for their equality

```
String result = myClass.concat("Hello", "World");  
assertEquals("HelloWorld", result);  
assertEquals("Reason for  
failure", "HelloWorld", result);
```

Will get printed if the test will fail

Note: All assert methods are static methods,
hence one has to use static import
import static org.junit.Assert.*;

Assert methods with JUnit (Contd.).

- **assertTrue() , assertFalse()**

- Used to test a variable to see if its value is true, or false

```
assertTrue (testClass.isSafe());
```

```
assertFalse (testClass.isSafe());
```

- **assertNull(),assertNotNull()**

- Used to test a variable to see if it is null or not null

```
assertNull (testClass.getObject());
```

```
assertNotNull (testClass.getObject());
```

- **assertSame() and assertNotSame()**

- Used to test if two object references point to the same object or not

```
String s1="Hello";
```

```
String s2="Hello";
```

```
assertSame(s1,s2); ->true
```

Annotations

- Fixtures

- The set of common resources or data that you need to run one or more tests

- @Before

- It is used to call the annotated function before running each of the tests

- @After

- It is used to call the annotated function after each test method

```
public class CalculatorTest {
    Calculator c=null;

    @Before
    public void before()
    {
        System.out.println("Before Test");
        c=new Calculator();
    }

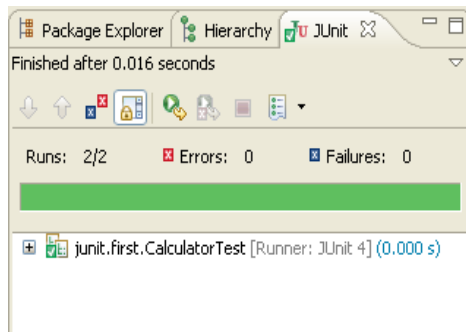
    @After
    public void after()
    {
        System.out.println("After Test");
    }
}
```

```
@Test
public void testAdd() {
    System.out.println("Add function");
    assertEquals("Result",5,c.add(2,3));
}

@Test
public void testSub() {
    System.out.println("Sub function");
    assertEquals("Result",20,c.sub(100,80));
}
}
```

O/P :

Before Test
Add function
After Test
Before Test
Sub function
After Test



Annotations (Contd.).

- **@BeforeClass**
 - The annotated method will run before executing any of the test method
 - The method has to be static
- **@AfterClass**
 - The annotated method will run after executing all the test methods
 - The method has to be static

O/P :
Before Test
Add function
Sub function
After Test

```
public class CalculatorTest {  
    static Calculator c=null;  
    @BeforeClass  
    public static void before()  
    {  
        System.out.println("Before Test");  
        c=new Calculator();  
    }  
  
    @AfterClass  
    public static void after()  
    {  
        System.out.println("After Test");  
    }  
  
    @Test  
    public void testAdd() {  
        System.out.println("Add function");  
        assertEquals("Result",5,c.add(2,3));  
    }  
    @Test  
    public void testSub() {  
        System.out.println("Sub function");  
        assertEquals("Result",20,c.sub(100,80));  
    }  
}
```

Annotations (Contd.).

- **@Ignore**

- Used for test cases you wanted to ignore
- A String parameter can be added to define the reason for ignorance

```
@Ignore("Not Ready to Run")
```

```
@Test
```

```
public void testComuteTax() { }
```

- **@Test**

- Used to identify that a method is a test method

Annotations (Contd.).

- **Timeout**

- It defines a timeout period in milliseconds with “timeout” parameter
- The test fails when the timeout period exceeds.

```
@Test (timeout = 1000)
public void testinfinity() {
while (true)
;
}
```

Parameterized test

Parameterised Tests

- New feature added in JUnit 4
- Used to test a method with varying number of Parameters
- Steps for testing a code with multiple parameters
 - The testing class should be annotated with `@RunWith(Parameterized.class)`
 - The class should have these 3 entities
 - A single constructor that stores the test data
 - Is expected to store each data set in the class fields
 - A static method that generates and returns test data
 - This should be annotated with `@Parameters`
 - It should return a Collection of Arrays
 - Each array represent the data to be used in a particular test run
 - Number of elements in an array should correspond to the number of elements in the constructor
 - Because each array element will be passed to the constructor for every run
 - A test method

Handling an Exception

- Two cases are there:
 - Case 1 :We expect a normal behavior and then no exceptions
 - Case 2: We expect an anomalous behavior and then an exception

Case 1:

```
@Test
public void testDiv()
{
    try
    {
        c.div(10,2);
        assertTrue(true); //OK
    }catch (ArithmeticException
    expected)
    {
        fail("Method should not
        fail");
    }
}
```

Case 2:

```
@Test
public void testDiv()
{
    try
    {
        c.div(10,0);
        fail("Method should fail");
    }catch (ArithmeticException
    expected)
    {
        assertTrue(true);
    }
}
```

```
public void div(int a,int b)throws ArithmeticException
{
    int c=0;

    c=a/b;
    System.out.println(c);}

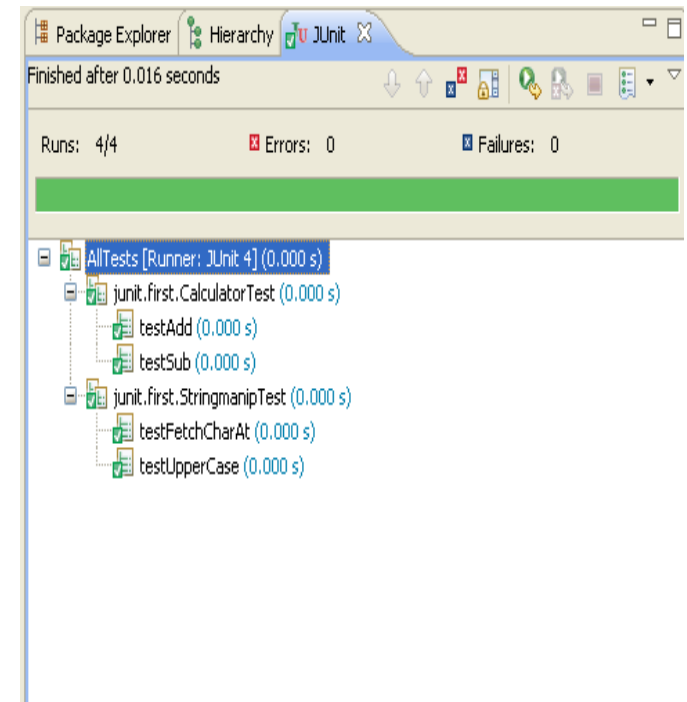
```

Test Suite

Test Suite

- Convenient way to group together tests that are related
- Used to bundle a few unit test cases and run it together
- Annotations used for this
 - `@RunWith`
 - Used to invoke the class which is annotated to run the tests in that class
 - `@Suite`
 - Allows you to manually build a suite containing tests from many classes

```
@RunWith(Suite.class)
@SuiteClasses({
    CalculatorTest.class,
    StringmanipTest.class
})
public class AllTests {
}
```



Quiz

1. Which of the following annotations has to be used before each of the test method?

- a. @Before
- b. @BeforeClass
- c. @After
- d. None of the above

None of the above

2. Which of the following are true?

- a. All assert methods are static methods
- b. The JUnit test methods can be private
- c. The JUnit test methods should start with the test keyword
- d. All of the above true

All assert methods are static methods

Summary

In this module, you were able to:

- Implement Naming Conventions for Java
- Implement Documentation Standards
- Implement Formatting Standards
- Implement Java Bean Conventions
- Explore JUnit?
- Learn about how to write Test cases?
- Implement the various assert methods
- Carry on Parameterized tests
- Understand Test Suites

References

1. Sun Microsystems. Java Coding Style Guide. Retrieved on April 10, 2012, from, <http://developers.sun.com/sunstudio/products>
2. Vincent Massol and Ted Husted. JUnit in Action. Ed 2. UK: Manning publications co., 2003.

Thank You