



Lesson Objectives

After completing this lesson, participants will be able to

- Understand importance of unit testing
- Install and use JUnit 4
- Use JUnit Within Eclipse



This lesson covers the one of the important development tool called JUnit. It explains how to work with JUnit, install, configure tests, run test with use of an IDE.

Lesson outline:

- 19.1: Introduction
- 19.2: JUnit
- 19.3: Installing and Running JUnit
- 19.4: Testing with JUnit
- 19.5: Testing Exceptions
- 19.6: Test Fixtures
- 19.7: Advanced Testing Concepts
- 19.8: Test Suites
- 19.9 :Parameterized Tests
- 19.10: Mocking Concept with EasyMock Framework

19.1: Introduction

Why is Testing Necessary



To test a program implies adding value to it.

- Testing means raising the reliability and quality of the program.
- One should not test to show that the program works rather that it does not work.
- Therefore testing is done with the intent of finding errors.

Testing is a costly activity.

Why Testing?

Why is Testing Necessary?

In SDLC, testing plays a vital role. When one tests a program one adds value to the program, in turn raising the quality and reliability of the program.

When we say “reliable”, it implies finding and removing errors. Hence one should not test a program to show that it works, but to show that program does not work.

Testing cannot guarantee against software problems or even failures but it can minimize the risks of faults developing once the software is put to use.

Typically when testing one should start with assumptions that the program contains errors and the test the program to find as many errors as possible.

Testing is a costly activity. A test which does not find an error is a waste of time and money. “A test case that finds an error is a valuable investment”.

19.1: Introduction

What is Unit Testing



The process of testing the individual subprograms, subroutines, or procedures to compare the function of the module to its specifications is called Unit Testing.

- Unit Testing is relatively inexpensive and an easy way to produce better code.
- Unit testing is done with the intent that a piece of code does what it is supposed to do.

Why Testing?

What is Unit Testing?

There are various phases in testing. However, in this course we are mainly concentrating on unit testing. Testing individual subprograms or procedure to compare the functions of the module to its required specifications is Unit Testing. This is an inexpensive activity and a very easy way to produce better code. In other words, unit test is a small piece of code that is written by the developer that will exercise a specific area of functionality of the code being tested.

For example: You write a functionality to sort a list and then check if the sort works. You then modify the functionality to accept the sort order, as well, and then you test this newly added functionality.

The point to note here is that while doing Unit testing, the developers are not worried about verification and validation of the program. It is just that the functionality should be running as required.

Unit Testing is also called as Test Driven Development (TDD). A significant advantage of TDD is that it enables you to take small steps while writing software. Most of you will be already doing some amount of unit testing in an ad hoc manner.

19.1: Introduction

What is Test-Driven Development (TDD)



Test-Driven Development, also called Test-First Development, is a technique in which you write unit tests before writing the application functionality.

- Tests are **non-production code** written in the same language as the application.
- Tests return a simple **pass** or **fail**, giving the developer immediate feedback.

Test Driven Development (TDD) requires developers to create automated unit tests that define code requirements before writing the code itself.

19.1: Introduction

Why Unit Testing



You can cite following reasons for doing a Unit Test:

- Unit testing helps developers find errors in code.
- It helps you write better code.
- Unit testing saves time later in the production/development cycle.
- Unit testing provides immediate feedback on the code.

Why Testing?

Why Unit Testing?

Why should a developer do unit testing? Some of the reasons that can be identified are given below:

Unit Testing helps developers find errors in code: When a developer starts writing unit tests, they can actually be surprised with how many errors are encountered in a small function that is written. It makes life of the developer easy. In case errors are “not found in time”, and are delayed till the end, then the entire module may fail.

Testing helps you write better code: Unit testing can help developer during the initial phase of development. Unit testing makes designs better and drastically reduce the time required while debugging.

Unit testing will save time later: To understand this concept, let us consider an example. Suppose you have written a small piece of code and you start testing. You identify that it is likely to fail in one tricky situation and the test finds out the error. You fix it, and continue the development. Since you have already fixed the bug, there is a little chance that the module might fail.

“Hence following the test driven development approach is beneficial”.

Unit testing provides immediate feedback on the code: When a developer is writing a critical module / functionality for a user interface, testing at that point will provide immediate feedback. There is no need to wait till the code becomes part of the application and then test it to check whether it works.

Overall Unit testing activity benefits the developer.

19.2: JUnit

Need for Testing Framework



Testing without a framework is mostly ad hoc.

Testing without framework is difficult to reproduce.

Unit testing framework provides the following advantages:

- It allows to organize and group multiple tests.
- It allows to invoke tests in simple steps.
- It clearly notifies if a test has passed or failed.
- It standardizes the way tests are written.

Why use JUnit?

Need for Testing Framework:

After having understood the need for Unit testing, there is a requirement to understand how to do Unit testing.

Mostly the Unit Testing done by the developers is ad hoc. The tests done in this manner are not put across in code at all. If at all they are put up, then they are written in such a manner that they cannot be reused in future. If they can be used in future, then typically they might be reproduced differently every time they are used. "Hence it is said that testing without a framework is difficult to reproduce".

With the help of a framework, the tests get documented in code and are reproduced in the same manner, whenever required.

19.2: JUnit What is JUnit



JUnit is a free, open source, software testing framework for Java.
It is a library put in a jar file.
It is not an automated testing tool.
JUnit tests are Java classes that contain one or more unit test methods.

Why use JUnit?

What is JUnit?

JUnit is an open source, software testing framework for Java developed by Kent Beck and Erich Gamma. JUnit allows developers to write Unit test cases for your Java code. It is library put in a jar file.

JUnit is not an automated testing tool. The developer has to write the test files and execute. JUnit offers some support so that the developer can easily write test files. It includes a tool which is called test runner to run your test files.

JUnit provides an easy way to state how the code should work. By expressing your intentions in code, you can use the JUnit test runners to verify that your code behaves according to your intentions.

19.2: JUnit Why JUnit



JUnit allows you to write tests faster while increasing quality and stability. It is simple, elegant, and inexpensive. The tests check their own result and provide feedback immediately. JUnit tests can be put together in a hierarchy of test suites. The tests are written in Java.

Why use JUnit?

Why JUnit?

Many developers will agree to the fact that the code should be tested before it is delivered. We have already seen the reasons why a testing framework should be used. However, it is also necessary to understand why JUnit should be used. Some of the reasons are defined below:

JUnit allows you to write tests faster while increasing quality and stability: Using JUnit, a developer spends less time debugging, and confidently makes changes to the code. With constant testing, any new functionality that is added can be verified for whether it is working or not. Hence the developer can be more positive about adding new features, because the developer now knows that it is less likely to fail. If a bug is detected while running tests, then the source code is fresh in your mind, so the bug is easily found. Also tests written in JUnit help you write code at a fast pace, and identify defects quickly. Writing tests builds the stability of the code and ensures that any changes that are made are working. As a result of the change, there is no effect on the software.

JUnit is simple, elegant, and inexpensive: "Simplicity" is the keyword while writing a test. Developers should not find it time consuming or difficult to write tests. With JUnit, the TDD can be followed very easily. It is simple and easy to put JUnit in practice. Developers can incrementally write tests as they increment their code. JUnit tests are such that they can be executed easily and frequently and it does not disturb the development process. This framework offers a cheap way of testing since it is an open source and freely downloadable ware.

JUnit tests check their own result and provide feedback immediately: Manual Unit testing is a tedious task and obviously it is time consuming to compare the expected and the actual result. As a result, developers tend to do away with Unit testing. JUnit tests can be run easily and they check their own results. The developer immediately gets a feedback if the tests have passed or failed. Hence any manual intervention is not required while executing the tests. JUnit tests can be put together in a hierarchy of test suites: JUnit tests can be organized into test suites containing test cases and even other test suites. The composite behavior of JUnit tests allows you to assemble collections of tests and automatically regression test the entire test suite in one go. You can also run the tests for any layer within the test suite hierarchy.

(Test Suites will be covered later in detail)

JUnit tests are written in Java: Testing Java software using Java tests forms a "seamless bond" between the test and the code under test. The tests become an extension to the overall software and code can be refactored from the tests into the software under test. The Java compiler helps the testing process by performing static syntax checking of the unit tests and ensuring that the software interface contracts are being obeyed.

19.3: Installing and Running JUnit

Steps for Installing JUnit



Following are the steps for installing and running JUnit:

- Download JUnit from www.junit.org. You can download either the jar file or the zip file.
 - Unzip the JUnit zip file
- Add the jar file to the CLASSPATH.
 - Set CLASSPATH=.,%CLASSPATH%;junit-4.3.1.jar

Installing and Running JUnit:

To start writing tests using JUnit, you will require the jar file for the latest version of JUnit. You can download the jar file or the zip file from the website. The zip file contains the source code and the documentation, as well.

Once downloaded, unzip the zip file that has downloaded. Add the jar file in the classpath. Alternatively the classpath can also be set through the Environment Variables option.

Once done you are ready to use JUnit.

19.3: Installing and Running JUnit Using JUnit within Eclipse

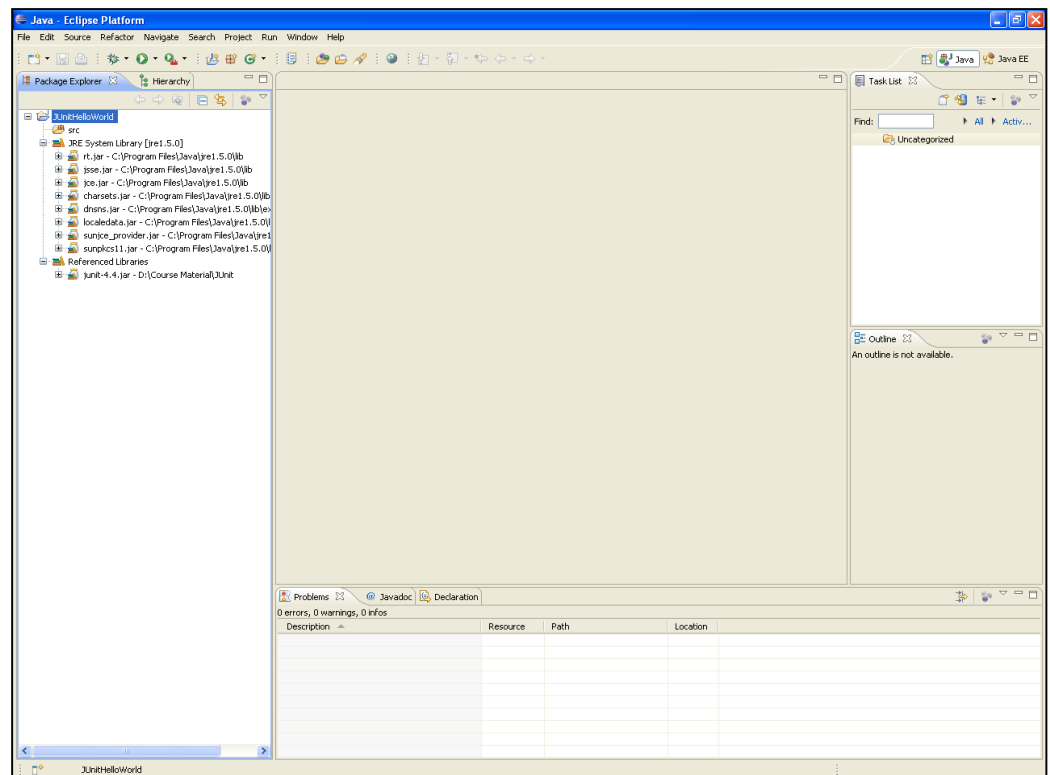
JUnit can be easily plugged in with Eclipse.
Let us understand how JUnit can be used within Eclipse.

- Consider a simple “Hello World” program.
- The code is tested using JUnit and Eclipse IDE.

Steps for using JUnit within JUnit:

- Open a new Java project.
- Add junit.jar in the project Build Path.

Note: The following figure depicts JUnit being added to the build path of the project.



19.3: Installing and Running JUnit

Using JUnit within Eclipse (Contd.)



Write the Test Case as follows:

```
import org.junit.Test;
import static org.junit.Assert.*;
public class TestHelloWorld {
    @Test
    public void testSay()
    {
        HelloWorld hi = new HelloWorld();
        assertEquals("Hello World!", hi.say());
    }
}
```

```
class HelloWorld{
    String say(){
        return "Hello World!"
    }
}
```

Note:

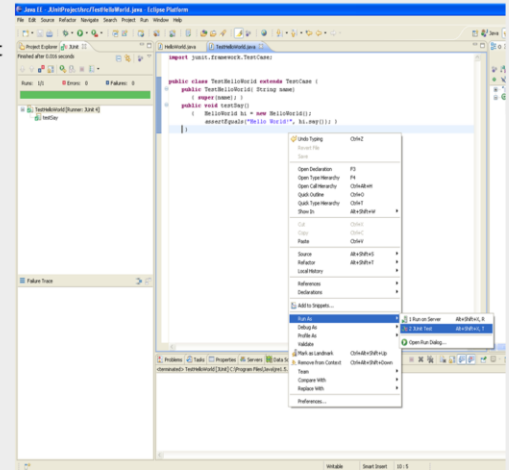
Test method must be annotated with @Test annotation then only Junit framework and eclipse will consider it as test method.

assertEquals(expected,actual) is performing the Test .This function internally uses Java assert feature which throws AssertionError exception when assertion fails.

19.3: Installing and Running JUnit Using JUnit within Eclipse (Contd.)

Run the Test Case.

- Right-click the Project → Run As → JUnit Test
- The output of the test case is seen in Eclipse.



19.3: Installing and Running JUnit

Demo

Demo on:

- Using JUnit with Eclipse
 - HelloWorld.java
 - TestHelloWorld.java



19.4: Testing with JUnit

Annotation Types in JUnit4.x



JUnit4.x introduces support for the following annotations:

- `@Test` – used to signify a method is a test method
- `@Before` – can do initialization task before each test run
- `@After` – cleanup task after each test is executed
- `@BeforeClass` – execute task before start of tests
- `@AfterClass` – execute cleanup task after all tests have completed
- `@Ignore` – to ignore the test method

Testing with JUnit – Annotations:

Annotation Types in JUnit4.x:

JUnit4.x introduced annotations. They are as follows:

`@Test` : It is used to signify a method as a test method. There are some options which can be mentioned with this annotation. For example:

`@Test(timeout=100)` fails if the test takes longer than 100 milliseconds for execution. This can be used to test infinite loops. Earlier you would have to start every method with 'test' which is now not required. The method can be named whatsoever. You have to simply prefix it with `@Test` annotation.

`@Before` : It is used to carry some task before each test is run. This can be used for initialization required before the test.

`@After` : It is used to perform cleanup after each test is executed.

`@BeforeClass` : It will execute the method before the start of the tests. This can be used for initialization of intensive resources like database connection.

`@AfterClass` : It will execute the method after all tests have finished. Cleanup activities are carried out.

`@Ignore` : It will ignore the test method. This can be useful when the base code has been changed but the test is yet to be revised. You can also use this annotation when you do not want to run a test currently because it takes too long.

(As we go ahead, we will understand each of the annotation in detail.)

19.4: JUnit Framework

Simple Example using JUnit4.x



Consider the following code snippet:

```
import static org.junit.Assert.*;
import org.junit.Test;
public class FirstJUnitTest {
    @Test
    public void simpleAdd() {
        int result = 1;
        int expected = 1;
        assertEquals(expected, actual);
    }
}
```

Understanding JUnit Framework:

Before we move any further, let us understand a very simple test case. Notice the static import. Since JUnit4.x is closely bound to Java 5, the static imports are permitted. JUnit4.x uses org.junit.* package, which provides JUnit core classes and annotations. As mentioned earlier, the TestCase class is not required to be inherited by the test class.

The class, which includes at least one @Test annotation, is treated as the test class. In JUnit4.x, the assert methods are static. Hence you need to do either of the following:

- Call assert methods using Assert.assertEquals().

- Do a static import as we have done in the snippet shown on the slide.

You also need not tag test methods by starting their name with "test". You instead only need to specify the @Test annotation.

19.4: Testing with JUnit

Assert Statements in JUnit



Following are the methods in Assert class :

- Fail(String)
- assertTrue(boolean)
- assertEquals([String message],expected,actual)
- assertNull([message],object)
- assertNotNull([message],object)
- assertEquals([String],expected,actual)
- assertEquals([String],expected,actual)
- assertEquals(String,T actual, Matcher<T> matcher)

Testing with JUnit – Assertions:

Assert Statements in JUnit:

The org.junit.Assert class provides a set of useful assertions methods. You can call these methods by either using Assert.assertEquals() or by referencing through static import. Any failed assertions will be only recorded. Some assertion methods are as follows:

Fail([String]) : It signals the failure of a test. This method has two formats. If the String argument is not provided, then no message is displayed. Else the String argument message is displayed.

assertTrue(boolean) : It asserts if the condition is true. Similarly

assertFalse(boolean) asserts if the condition is false.

assertEquals([String message],expected,actual) : It asserts whether the two objects passed as arguments are equal. This method can accept any kind of values for comparison like double long, etc..

assertNull([message],object) : It asserts that an object is null.

assertNotNull([message],object) : It asserts that an object is not null.

assertSame([String],expected,actual) : It asserts that two objects refer to the same object and assertNotSame([String],expected,actual) asserts that two objects do not refer to the same object.

assertThat(String, T actual, Matcher <T> matcher) : It asserts that “actual” satisfies the condition specified by the “matcher”.

19.4: Testing with JUnit Demo



Demo on:

- Using @Test Annotation
- Using Assert Methods
 - Counter.java & Testcounter.java
 - Person.java & TestPerson.java



Note:

Refer to JUnitProject.

Demo 1: Counter.java and testCounter.java

Demo 2: Person.java and TestPerson.java

In both the examples, we have used the @Test annotation and the assert methods to evaluate if the methods in the underlying class are working properly.

Remove the @Test annotation from one of the Test methods, and then observe the output. That particular method is not considered as a test method.

19.4: Testing with JUnit

Using @Before and @After



Test fixtures help in avoiding redundant code when several methods share the same initialization and cleanup code.

Methods can be annotated with @Before and @After.

- @Before: This method executes before every test.
- @After: This method executes after every test.

Any number of @Before and @After methods can exist.

They can inherit the methods annotated with @Before and @After.

Testing with JUnit – Test Fixtures:

Using @Before and @After:

Many a times a set of common resources or data that you need to run one or more tests are required. To avoid the redundant code when several methods share the same initialization and cleanup code, you can use @Before and @After annotations. Prefix the methods with the respective annotations. In the previous version, you had to use setup() and teardown() methods to perform this task.

@Before annotated methods run before every test, and @After prefixed methods run after every test.

You can also have any number of @Before and @After prefixed methods as you need. It is possible to inherit the @Before and @After methods. The @Before methods in the superclass will be executed prior to the derived class @Before methods. The @After in the subclass are executed before the superclass @After methods. The superclass @Before and @After methods execute automatically and there is no need to call them explicitly. Also a logger can be initialized in @Before method of a Test Case.

When inherited, the overall execution process will be as follows:

- @Before methods in the superclass
- @Before methods in the current class
- @Test methods in the current class
- @After methods in the current class
- @After methods in the superclass

19.4: Testing with JUnit Using @Before and @After



Example of @Before:

```
@Before
public void beforeEachTest() {
    Calculator cal=new Calculator();
    Calculator cal1=new Calculator("5", "2"); }
```

Example of @After:

```
@After
public void afterEachTest() {
    Calculator cal=null;
    Calculator cal1=null; }
```

Note: The methods annotated with @Before and @After should be declared as public void.

19.4: Testing with JUnit Demo



Demo on:

- Using the @Before and @After annotations
 - TestPersonFixture.java



Note:

Demo 5: TestPersonFixture.java

Notice the output of the execution of @Before and @After methods.

19.5: Testing Exceptions

Testing Exceptions



It is ideal to check that exceptions are thrown correctly by methods. Use the expected parameter in @Test annotation to test the exception that should be thrown.

For example:

```
@Test(expected = ArithmeticException.class)
public void divideByZeroTest() {
    calobj.divide(15,0);
}
```

Testing with JUnit – Testing Exceptions:

Testing using the Exceptions:

At times you would want to check that an exception is thrown correctly under certain circumstances. In previous versions, it was a laborious task which involved writing a try block around the code that throws the exception, then adding a fail() statement at the end of the try block.

For example:

```
public void TestDivideByZero()
{
    try {
        calobj.divide( 15, 0 );
    }
    catch (ArithmeticException e) {
        fail(e.getMessage);
    }
}
```

In this case, if the exception is not thrown or a different exception is thrown, then the test will fail.

Now, you can use the expected parameter in the @Test annotation to test the exception that should be thrown as shown on the slide.

However, you may still need to use the traditional try-catch block if you want to test the exception's detail message or other properties.

19.5: Testing Exceptions
Demo

Demo on:

- Exception Testing
 - Person.java & TestPerson2.java



19.6: Test Fixtures

Using @BeforeClass and @AfterClass



Suppose some initialization has to be done and several tests have to be executed before the cleanup.

Then methods can be annotated by using the @BeforeClass and @AfterClass.

- @BeforeClass: It is executed once before the test methods.
- @AfterClass: It is executed once after all the tests have executed.

Testing with JUnit – Test Fixtures:

Using @BeforeClass and @AfterClass:

Sometimes the requirement can be wherein the initialization code is run, and then several tests are executed. In this scenario, we need to use @BeforeClass and @AfterClass annotated methods. This can also be termed as “one time setup and teardown”.

@BeforeClass : The method annotated with this will be executed once before the tests are run. It means if you have just one test or ten tests, then this will run one time before the very first test executed. In case of inheritance, the base class @BeforeClass method will execute once.

@Afterclass : The method annotated with this will run once after all the tests have finished execution. In case of inheritance, the @AfterClass in the derived class will execute first and then the method from base class will be executed.

Unlike @Before and @After, only one set of @BeforeClass and @AfterClass annotated methods are allowed.

19.6: Test Fixtures

Using @BeforeClass and @AfterClass



Example of @BeforeClass:

```
@BeforeClass
public static void beforeAllTests() {
    Connection conn=DriverManager.getConnection(...);}
}
```

Example of @AfterClass:

```
@AfterClass
public static void afterAllTests() {
    conn.close; }
}
```

The methods using this annotation should be public static void

The methods using this annotation should be public static void. Because @BeforeClass gets called before the class is created and @Afterclass complements the @BeforeClass annotated method.

19.6: Test Fixtures
Demo



Demo on:

- Using the `@BeforeClass` and `@AfterClass` annotations



Note:
Demo 6: TestPersonFixture.java

19.6: Test Fixtures Using @Ignore



The @Ignore annotation notifies the runner to ignore a test. The runner reports that the test was not run. Optionally, a message can be included to indicate why the test should be ignored. This annotation should be added either in before or after the @Test annotation.

Testing with JUnit – Ignoring Tests:

Using @Ignore:

In JUnit4.x, if you need to temporarily ignore a test from being executed, then annotate them with @Ignore.

The @Ignore annotation can be included either in front or after the @Test annotation. The runner then ignores these tests and reports how many tests were not run along with the pass and fail tests.

Optionally, a message can be included to indicate why a test should be ignored for a particular test that is run. Though it is optional it is recommended that the message should be given because you might forget why a particular test has been ignored

In addition to this, even a class can be annotated with @Ignore, and all the tests in that class will be ignored.

The @Ignore annotation is should ideally be used when

the method cannot be tested in some form and it is documented in the code. This should be a special case and warrant a discussion or code review to see if there is any way to test it.

The test is not yet built - this should ideally happen only for legacy code.

This should be also subject to code review and tasks should be put to add tests.

19.6: Test Fixtures Using @Ignore



Example of @Ignore for a method:

```
@Ignore ("The network resource is not currently available")  
@Test  
public void multiplyTest() {  
    .....  
}
```

Example of @Ignore for a class:


```
@Ignore public class TestCal {  
    @Test public void addTest(){ .... }  
    @Test public void subtractTest(){.....}  
}
```

19.6: Test Fixtures

Demo

Demo on:

- Using the @Ignore
 - Student.java & TestStudent.java



Note:

Demo 7: TestStudent.java and Student.java

The Ignore annotation included in the program will not allow the respective test case to execute. The JUnit panel also indicates that one test has been ignored.

19.7: Test Suites

Composing Test into Test Suites



Example:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({ TestCalAdd.class,
TestCalSubtract.class,
TestCalMultiply.class, TestCalDivide.class })
public class CalSuite {
// the class remains completely empty,
// being used only as a holder for the above annotations
}
```

Testing with JUnit – Test Suites:

Composing Test into Test Suites:

JUnit provides you with (contd.):


@RunWith : JUnit invokes the class it references to run the tests when this annotation is used instead of the runner that is built into JUnit. In JUnit4.x, suites are built using **@RunWith** and a custom runner named Suite. The **@RunWith** is telling JUnit to use `org.junit.runner.Suite` class. This runner allows you to manually build suite containing tests from many classes. All classes are defined in **@Suite.SuiteClasses**

@Suite.SuiteClasses : This annotation is used to specify an array of test classes for the runner.

The CalSuite class is simply a place holder for the suite annotations. The


@RunWith is the annotation which tells the JUnit runner to use the `org.junit.runner.Suite` class for running a particular class. The **@Suite** annotation instructs the suite runner about the test classes which have to be included in this suite and the sequence in which they should be introduced.

19.7: Test Suites
Demo



Demo on:

- Composing tests into Test Suites
 - TestPersonSuite.java



Note:

TestPersonSuite.java

In this demo example, the three classes, namely TestPerson, TestPerson2, and TestPersonFixture, have been put together for execution. The class itself does not have any methods for testing.

19.8: Parameterized Tests

Reusing Tests



Parameterized tests allow you to run the same test with different data.

To specify parameterized tests:

- Annotate class with `@RunWith(Parameterized.class)`.
- Add a public static method that returns a Collection of data.
 - Each element of the collection must be an Array of the various parameters used for the test.
- Add a public constructor that uses the parameters.

Testing with JUnit – Parameterized Tests:

Reusing Tests:

One of the significant features in JUnit4.x is the ability to run parameterized tests. This feature allows you to run the same test with different data. Essentially it means that you create a generic test, and run it multiple times with different parameters. To create parameterized tests, use the specification as given on the slide.

19.8: Parameterized Tests Reusing Tests



Example:

```
@RunWith(Parameterized.class)
public class SomethingTest {
    @Parameters
    public static Collection<Object[]> data() { .... }
    public SomethingTest()
    {.....}
    @Test
    public void testValue()
    {.....}
}
```

Testing with JUnit – Parameterized Tests:

Reusing Tests:

As per the syntax given on the slide, to create parameterized tests annotate:

- The class with `@RunWith`

- The method, which returns the collection to be annotated, with

- `@Parameters`

19.4: Mocking Concepts

Testing in Isolation



Unit Testing of any method should be ideally done in isolation from other methods.

For testing in isolation, you need to be independent of expensive resources.

Use mock objects to perform testing in isolation.

Mock object is created to represent an object that your code will be collaborating with.

Testing with JUnit – Isolated Testing:

Testing in Isolation:

Unit testing any method should ideally be done in isolation from other methods and it is certainly a nice objective. However, testing in isolation can get difficult in certain scenarios.

For example: The business logic of an application is built into servlet and without running the server you cannot use the functionality.

Hence you need an object that can be used in a test instead of an expensive resource or a difficult resource. That is instead of using a real database, we can use a mock object representing the database. The usage of mock objects allows you to unit test at a fine grained level by allowing you to write unit tests for all methods.

Hence while defining a mock object we can say, “a mock object is created to represent an object that your code will be collaborating with”.

19.4: Mocking Concepts

Advantages of Using Mock Objects



There are obvious advantages of using mock objects:

- You get the ability to test code that is not yet written.
- They help teams to unit test one part of the code independently.
- They help to write focused tests that will test only a single method.
- They are helpful when the application integrates with expensive external resources.

Testing with JUnit – Isolated Testing:

Advantages of Using Mock Objects:

There are some noticeable benefits of using mock objects:

You can test the code which is not yet written but at the minimum an interface should be available to work with.

Testing in isolation helps teams to test one part of the code independently without waiting for all other parts of the code.

Also you can write focus tests that test only a single method without side effects resulting from other objects that are called from the method. Small focused tests are easy to understand and they do not break when other parts of the code are changed.

Mock Objects replace the objects with which your method under test collaborates, thus offering a layer of isolation.

Mock objects are quite helpful to write a test wherein that part of the code integrates with expensive external resources.

19.4: Mocking Concepts

Mock Objects in JUnit



Mock objects can either program these classes manually or use EasyMock to simulate these classes.

- EasyMock provides mock objects for interfaces in JUnit tests.
- EasyMock is an open source software that is available under the terms of the Apache 2 license.

Testing with JUnit – Isolated Testing:

Mock Objects in JUnit:

Other mock frameworks available are DynaMock and JMock.

In this course, we have a simple demo to understand representation of mock objects using EasyMock. Here we do not go into much details of EasyMock.


However, you need easymock.jar in your classpath.

19.4: Mocking Concepts

Demo

Demo on:

- Using Mock Object in JUnit
 - demo.mock.LoginTest.java



Note:

Refer to demo.mock package for this demo

The UserDao functionality has to be tested. No concrete implementation of this interface exist. We use a mock object to check the functionality of the method in UserDao

The test depends on the provided methods.

The expect method tells EasyMock to expect certain method with some arguments and return method defines the return value of this method.

The replay method needs to be called to make mock objects available.

The verify method tells EasyMock to validate that all expected method calls were executed and in the correct order

19.7: Best Practices Unit Testing



Start with writing tests for methods having the fewest dependencies and then work your way up.
Ensure that tests are simple, preferably with no decision making.
Use constant, expected values in the assertions instead of computed values wherever possible.

Best Practices in JUnit:

Unit Testing Best Practices:

Unit Testing is the execution and validation of a block of code, which is created by a developer, in isolation. Some best practices to keep in mind while doing unit testing with JUnit are elaborated as follows:

Start with writing tests for methods having the fewest dependencies and then work your way up. If the testing process starts from a higher level method, then there is a probability that the test may fail since the subordinate method returns an incorrect value to the high level method. This increases the time required to find the cause of the problem, and to do this you need to test the subordinate method. Hence start with a bottom up approach

Ensure that tests are simple with no decision making. Decision making in the tests add to the number of ways a test method can be executed.

Normally the requirement is to change the input of the method only.

Use constant expected values in the assertions instead of computed values wherever possible. Consider the following example:

```
returnVal=methodToTest(input);  
assertEquals(returnVal,computeExpected(input));  
assertEquals(returnVal,12);
```

In this snippet, the computeExpected method will have to implement the similar logic as the method which is being tested. The test class will tend to grow lengthy. Moreover, the second assertion statement is easy to implement, understand, and maintain.

19.7: Best Practices Unit Testing



Ensure that each unit test is independent of all other tests.
Clearly document all the tests.
Test all methods whether public, protected, or private.
Test the exceptions.

Best Practices in JUnit:

Unit Testing Best Practices:

Ensure that each unit test is independent of all other tests. The test method ideally executes one specific behavior for a single method. Placing too many assertions in one test case may cause a problem. This is because even if one of the assertion fails, then the entire test fails. Strive for one assertion per test case.

Clearly document each unit test. The name of the test method should ideally indicate which method is being tested. This helps in easy maintenance and reduced efforts in refactoring. Provide proper supporting comments to describe any special conditions.

Test all methods whether public, protected, or private.

Create unit tests which specifically check for exceptions. If a method throws more than one exception, then appropriate unit tests must be created to simulate those situations when the exceptions will be thrown.

Note: Testing exceptions will be discussed in Lesson 17.

19.7: Best Practices JUnit



Do not use the constructor of test case to setup a test case, instead use an `@Before` annotated method.
Do not assume the order in which tests within a test case should run.
Place tests and the source code in the same location.
Put non-parameterized tests in a separate class.

Best Practices in JUnit:

JUnit Best Practices:

Some more best practices to be followed while working with JUnit are elaborated as follows:

Do not use the constructor of test case to setup a test case, instead use an `@Before` method to do the setup task. The reason for this is in case the constructor fails to do the setup, JUnit simply throws an `AssertionFailedError` which indicates that the test could not be instantiated. Also the error stack trace is not very informative. As a result, it makes it hard to figure out the exceptions underlying cause. Using the `@Before` annotated method is handy because any exception thrown within this method is reported correctly and the error stack trace is informative. Hence tracking the probable cause of error is easy.

Never assume that tests will be called in a specific order. If the tests are dependent on each other, then it is better to compose them in test suites. In this way, it is controlled that the tests run in the order in which they were mentioned.

Keep both the tests and source code in the same location. This will compile both test and class during a build. In this case, the tests and class are synchronized during the development.

Put non-parameterized tests in a separate class to avoid running of that test since parameterized tests create new instance of the class every time. This leads to running the non-parameterized tests also every time.

19.7: Best Practices

JUnit



When writing tests consider the following questions:

- When do I write tests?
- Do I test everything?
- How often the tests should be run?
- Why use JUnit, instead why not use `println()` or a debugger?

Best Practices in JUnit:

JUnit Best Practices:

While writing test cases using JUnit, you should consider certain important aspects:

When do I write tests?

Tests should be ideally written before the code. Follow the test driven development approach instead of keeping the test to be written at the very end. The bugs can then be corrected immediately as they are reported.

Do I test everything?

At least test everything that can possibly break. Maximize the testing investment since it is equivalent to investing in design. If something is difficult to test, then relook at your design. Improve the design so that it is easier to test.

How often the tests should be run?

Run all your unit tests as often as possible that is whenever there is a change in code. Running the test cases often gives the confidence that the changes did not cause anything to break or fail.

Why use JUnit, instead why not use `println()` or a debugger?

Inserting `println` or debugging statements into the code causes the code to become lengthy. Also you need to manually scan the output every time your program runs to ensure that the program does what it is expected to do. Using debugger is a manual process that also requires visual inspections. It takes less time in the long run to codify expectations in the form of an automated JUnit test that retains its value over time.

Lab : Introduction to Junit



Lab 10: Introduction to Junit



Review Question

Question 1: Why should one do Unit Testing?

- Option 1: Helps to write code better
- Option 2: Provides immediate feedback on the code
- Option 3: Because it is one of the testing methods that has to be carried out

Question 2: JUnit is a licensed product and can be purchased with Java.

- True / False



Review Question

Question 3: To start working with JUnit in Eclipse, you need to add junit.jar in ____.

- Option 1: CLASSPATH
- Option 2: BUILDPATH
- Option 3: Project Settings

Question 4: You can add a number of tests using the <batchtest> element.

- True / False

