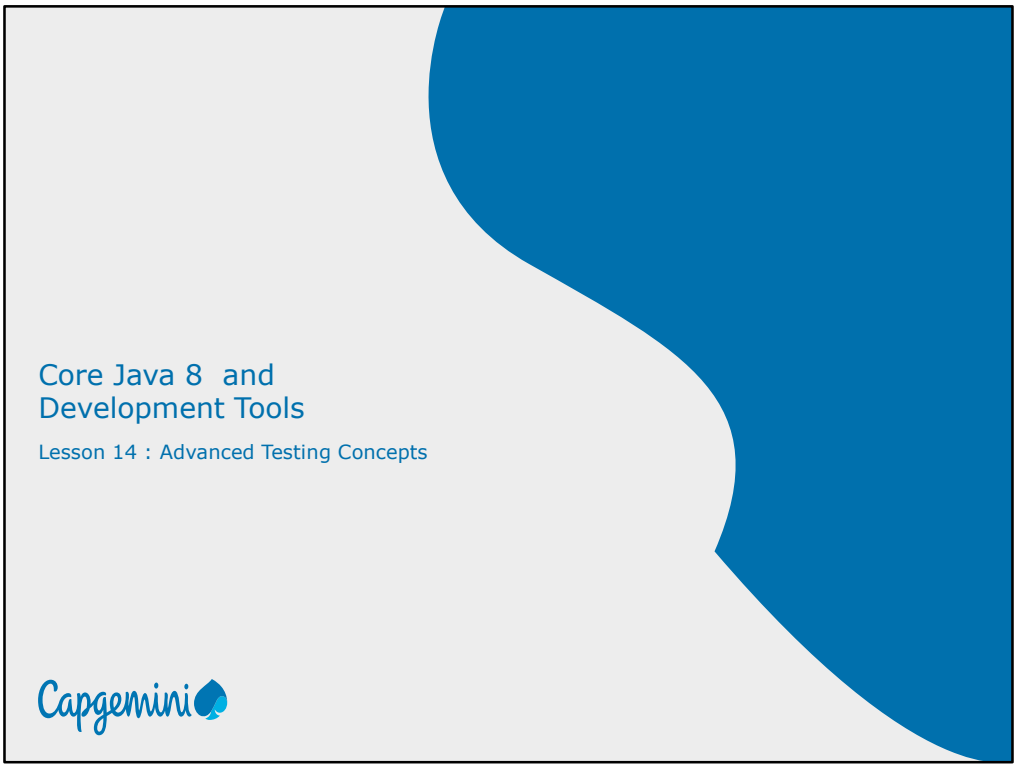
Core Java 8 and
Development Tools
Lesson 14 : Advanced Testing Concepts
Capgemini

## Lesson Objectives

After completing this lesson, participants will be able to

- Understand advanced testing concepts
- Work with test suites
- Implement parameterized tests and mocking concepts

This lesson covers advanced testing concepts.

Lesson outline:

20.1: Advanced Testing Concetps
20.2: Best practices

14.1: Advanced Testing Concepts

## Composing Test into Test Suites

A testsuite comprises of multiple tests and is a convenient way to group the tests, which are related.

It also helps in specifying the order for executing the tests.

JUnit provides the following:

- org.junit.runners.Suite class : It runs a group of test cases.
- @RunWith : It specifies runner class to run the annotated class.
- @Suite.SuiteClasses : It specifies an array of test classes for the Suite.Class to run.
  - The annotated class should be an empty class but may contain initialization and cleanup code.

Testing with JUnit – Test Suites:
Composing Test into Test Suites:
In a software development environment, a collection of test cases that test a software program is called a test suite. The tests in the test suite are normally related. For example: All the tests are testing for mathematical functionality. The test suite runs a collection of test cases. Prior to this version, you create a test suite using an instance of junit.framework.TestSuite. So the code looks as follows:

```
public static TestSuite()
{
   TestSuite RunTests = new TestSuite();
   RunTests.addTest(new MyTest("testFirstMethod"));
   RunTests.addTest(new MyTest("testSecondMethod"));
   return RunTests;
 }
```

However, the current version of JUnit encourages you to make use of annotations that have been introduced to build a Test Suite.
JUnit provides you with:

org.junit.runners.Suite : This class runs a group of test classes. Using this class as a runner lets you manually build a suite containing tests from several classes. To use it, annotate a class with @RunWith and @SuiteClasses.

14.2: 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.

14.2: 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.

14.3: 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.

14.3: 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

14.4: 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”.

14.4: 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.

14.4: 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.

14.4: Mocking Concepts

## Demo

**Lab-5**

- Using Mock Object in JUnit


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

## Summary

In this lesson, you have learnt:

- Advanced Testing Concepts

## Review Question

Question 1: While writing unit tests you should test ___.

- Option 1: Only public methods
- Option 2: Only constructors
- Option 3: Should test all methods

Question 2: Use constant expected values in assertions.

- True / False