**Java Unit testing with JUNIT and Mockito**

Course offering-

What is Unit testing?

What is Junit?

UnitTesting with Junit and Mockito.

How to create testable applications?

**GitHub repo link-** [GitHub - in28minutes/MockitoTutorialForBeginners: Mockito Tutorial for Beginners](https://github.com/in28minutes/MockitoTutorialForBeginners)

**Introduction-** Unit testing is when you try and test a single unit.

A unit can be a method, group of method, classes or set of classes.

Some conventions-

* Keep test cases code separated from source code.
* Test class should be in same package as the source class.
* Method should be always public void. It should never return anything.
* @Test annotation. Annotation came into picture with java 1.5 before that there were no annotations. So, we used method as test\_methodName.
* The name of test class should be SourceClassNameTest.

As StringHelper class test class name should be StringHelperTest

Good coding practices-

* Don’t use System.out.println in code, preferably set up logger and use that.

**@Before Annotation-**

If we require some setup to be done before running test as passing some data, then run a @Before test. It is run before every test.

**@After Annotation-**

If we require some setup to be done before running test as passing some data, then run a @After test. It is run after every test.

**@BeforeClass Annotation-**

If we require some setup to be done before running test as passing some data then run a @BeforeClass test. It is run before every test.

The method implementing @BeforeClass annotation should be static. So all constraint around static method are there. Cannot refer to instance variable. Only class level can be initialized.

**@AfterClass Annotation-**

If we require some setup to be done before running test as passing some data, then run a @AfterClass test. It is run after every test. Constraint on @BeforeClass appliable here too.

***AssertEquals***

assertEquals doesn’t check for equality rather it checks for if objects are same.

So, for array comparison, use assertArrayEquals.

***Exceptions***

@Test offers expected value to have exception that you expect to occur.

***Parameterized test***

In complex applications, we will need to create tests which are parameterized.

So, we use annotation @RunWith (Parameterized.class)

We also have an option to run multiple test classes with Suite. Suite.class offers to sun multiple test classes it is used with RunWith annotation. We can pass list of test classes we want to run.

**Mocking with Mockito**

What is stub?

We have a TodoService interface with a method retrieveTodos(String user) and a TodoBusinessImpl which implements method retrieveTodosRelatedToSpring(String user). Now we don’t have actual implementation of retrieveTodos but for testing method we will create a stub.

A stub is used for unit testing.

So, lets first create TodoServiceStub and then use that to Test TodoBusinessImpl.

What is Mocking?

Mocking is creating objects that simulates behavior of real objects, unlike stubs, mock can be dynamically created from code at runtime. Mocks tend to offer more functionality than stubbing. You can verify method calls.

mock is a method defined in Mockito class.This offers to create mock of a class or interface.

Doing complex thins with stub is difficult while mock method makes is very easy. We used two methods one mock for creating mock of class and other when and thenReturn is used to stub the mock.

***Mocking***

Mock List interface —Mockito provides a better mocking practice as if you invoke a method for the expectation has not been set then by default you get a null value. While this fault tolerance was not part of ***EasyMock*** a very famous mocking framework.

During mocking list interface, we mocked method like list.size and list.get(0) using mock and when\_thenReturn method from Mockito class.

Now if we want to get specific and want to return same output irrespective of index in list.get(index) method. There comes role of argument matcher.

***Introduction to BDDs-***

Behavior driven development is where we create user stories and start splitting them into scenarios. Scenarios are written in given when then format.

This is what we are going to do with unit testing

//Given //When //Then

Mockito provides us with specific class BDDMockito which allows us to do so.

***Verify calls on mocks***

Verify is useful for methods which return void as there we can not use assert methods. Verify helps us identify if method was called.

verify(mockService).method(parameter)

Same verify can be done with Bdd with then\_should also.

*then(mockService).should().method(parameter)*

***Hamcrest Matchers***

Hamcrest matchers provide us with a variety of methods that are much more readable and easily accessible. Methods like hasSize, arrayContaining for array make it easier to test code.

Note-Hamcrest core is compiled with junit dependency, adding explicitly hamcrest core and library dependency leads to error.

*java.lang.NoSuchMethodError: org.hamcrest.Matcher.describeMismatch(Ljava/lang/Object;Lorg/hamcrest/Description;)at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:18)*

Which can be resolved by excluding hamcrest dependencies from Junit.

***Java Annotations for mocking***

**@Mock –**

To create mock for a class.

We need to RunWith (MockitoJUnitRunner.class)

**@InjectMocks –**

To create instance and pass appropriate mocks.

**@Captor –**

To create argument captor

***Rules-***

We can not a class with multiple JunitRunners to overcome this Mockito brought rules.

@Rule annotation was introduced in Junit 4.7. Rule must be always declared as public.

Junit is now moving away from JunitRunners.