**Mockito Notes by Dinesh Varyani**

**What is Unit Test?**

Unit Test is a piece of code written by developer to test certain functionality of code. The class which we want to test is called as **Class Under Test**. The Method which we want to test is called as **Method Under Test**.

A unit Test instantiate class under test, execute method under test and verifies that it worked as expected.

**A good unit test should be**

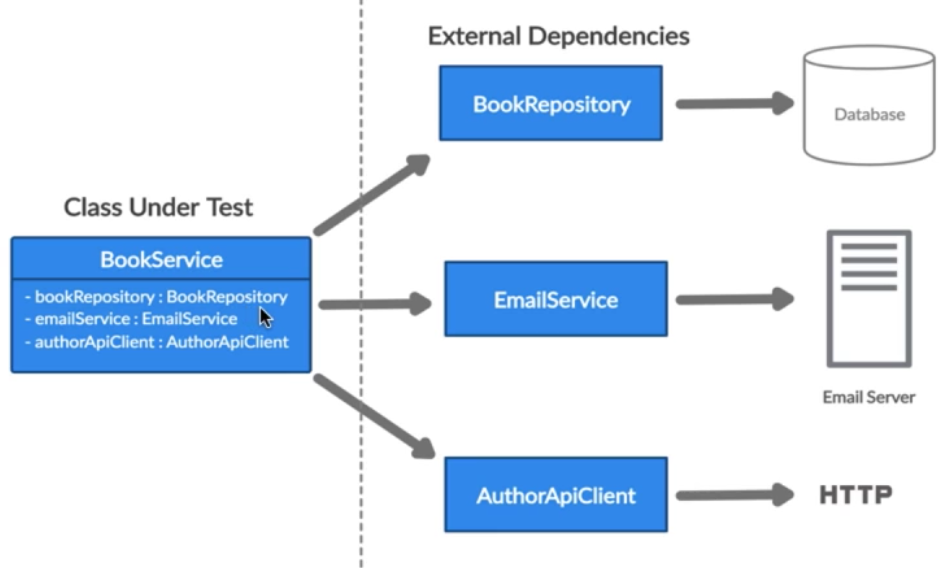
a. Easy to Test.

b. Easy to Read.

c. Reliable. (it should not happen that one unit test is running on one machine and failing on another)

d. Fast.

e. Isolated. (Class Under Test should not be dependent on network calls, database calls, any services, or other external dependencies. All these external dependencies should be removed from unit tests by replacing real objects with their fake replacements called as test doubles).



**What are Test Doubles?**

It is generic term for any case where we replace a production object for testing purpose.

We need test double because sometime it is not possible to unit test a code because of the unavailability of external dependency.

For Example:

if code under test is dependent on database operations, it is not possible to unit test the code unless the database is available. **or**

if code under test is dependent on external services, it is not possible to unit test the code unless the network is available. **or**

if code under test is sending an email, it is not good to send email every time test are run.

**Different Types of test doubles.**

a. Dummy

b. Fake

c. Stub

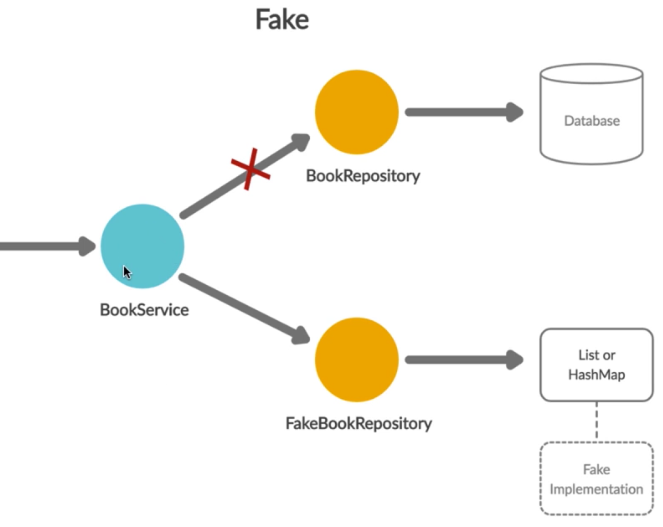
d. Mock

e. Spy

**Course Prerequisites** 🡪 Create a new Spring Boot Application.

**Fake Test Doubles**

Instead of actual object(production object) we use fake objects, Fake objects have working implementation, but not as production objects.



For Example: Instead of actual production database, we can use an in memory database, hash map or list.

**Dummy Test Doubles**

When we do unit testing for class under test, class may be dependent on external dependencies and it may happen that when we are testing one of the methods for class under test one or two dependencies does not play any role.

So these 2 dependencies do not play any role but still they come into picture. so while unit testing we create dummy test double out of it.

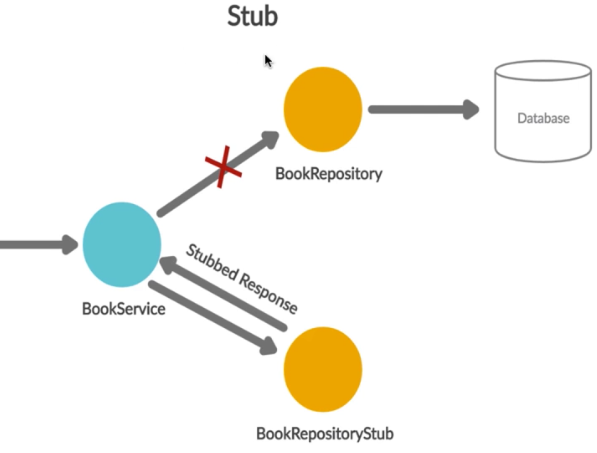
Dummy Test Double does not have any business logic. Dummy Test doubles are used only for code to compile.

**Stub Test Double**

Stub Test Doubles are predefined answers (hardcoded answers) to method executions made during the test.

In larger picture Stub Test double act as a replacement for our external dependency and whatever method is executed on external dependency,

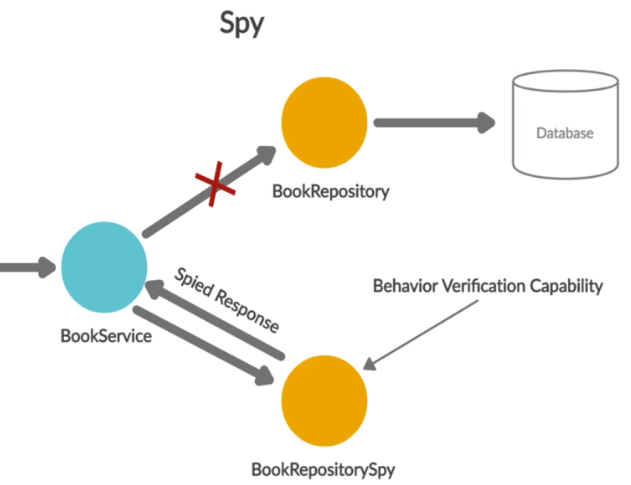
Instead of calling that external dependency the stub is called and some of the code is hardcoded in that particular method which does something and finally returns us back expected response.

****

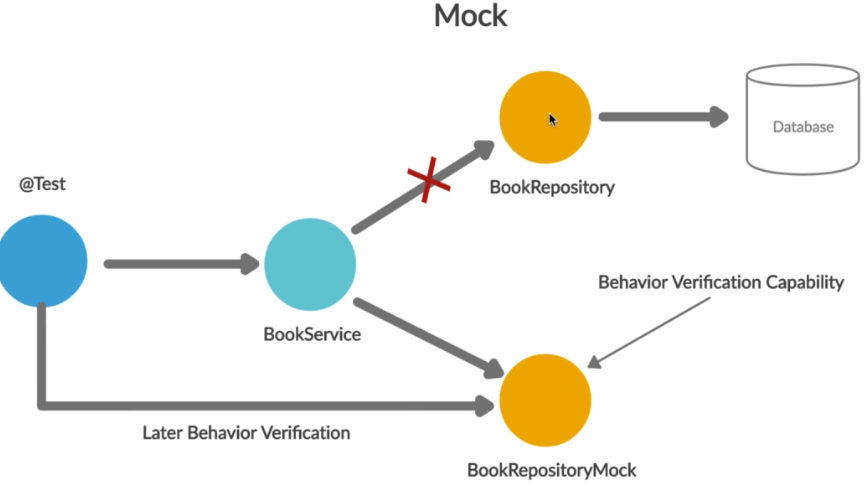
**Spy Test Double**

Spy Test Double keeps an eye on external dependency and it records what all interactions have been made with external dependency.

Spy Objects are very similar to Stub, but they record information about how they are executed.

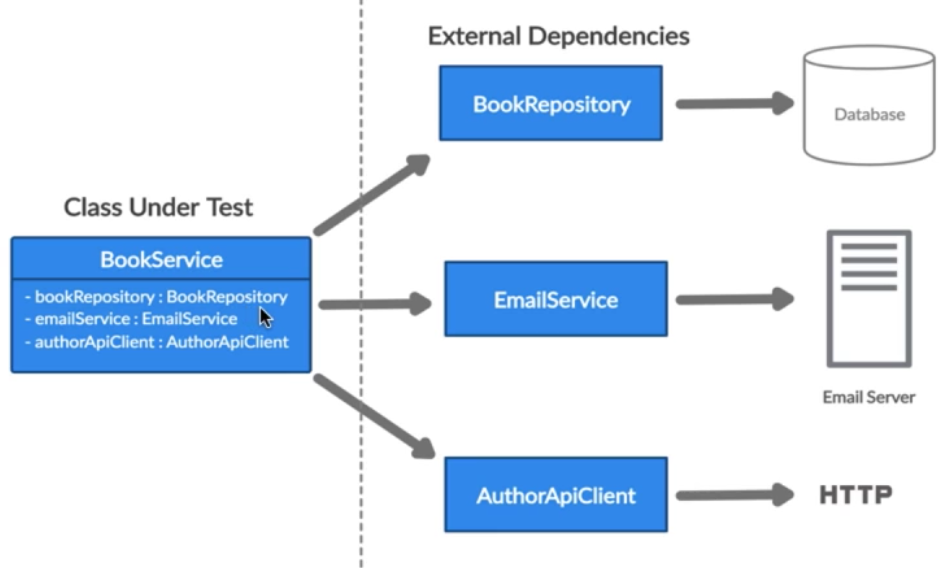


**Mock Test Double**

It is very similar to spy....with the only difference that mock object records method calls on it, and verifies later that the recorded calls match and assert and behavior verification is done at mock object. 

**Mocking**

Mocking helps us in testing functionality of class in isolation.



**Advantages of Mocking**

1. It does not require database connection, communicating to external servers or going over network to test functionality.

2. Mock Objects simulate real service by tasking in actual input and returning back expected output.

Mockito is testing framework for java.

It allows us to creation of test doubles objects (mock objects) for unit tests.

It uses Java reflection in order to create mock objects.

**Benefits of Mockito**

1. Save developers from writing test doubles or mock objects on their own.

2. Provide support for unit testing.

3. Help us in returning configured response.

4. It can check order of method called and number of time a method being called.

5. Annotation Support.

**Annotation Support in Mockito**

@Mock 🡪 It is used over the classes which we want to mock.(In laymen term it is used to identify External dependencies)

@ExtendedWith(MockitoExtension.class) 🡪 This annotation tells Mockito that we will be using @Mock Annotation. It is applicable in JUnit 5 Annotation

@RunWith(MockitoJUnitRunner.class) 🡪 It is used in JUNIT 4. . It tells Mockito that we will be using @Mock Annotation.

@InjectMocks🡪 it is used for identifying class under test.

There are bunch of other Annotations used in JUnit 4.

**What is Stubbing**

One of the primary benefits of Mockito is ability to return a provided response when a specific method is called on mocked dependency.

The process of writing how a given mock method should behave is called stubbing.

The feature is performed in 2 ways:-

1. Using Mockito Static method “*when() + thenReturn()*”. It states that when any specific method is called on Mock object, then return preconfigured value.
2. Using Mockito Static method “*doReturn() + when()*”. It states that ‘doReturn’ preconfigured value, ‘when’ specific method is called on mock object.

**Tips for Stubbing in Mockito**

1. Mockito uses equals() method while matching arguments during stubbing.
2. In case response is not stubbed for a method, default values are returned when called.

Default value of primitive is zero.

Default value of Boolean is false.

Default value of Object and Array is null.

Default value of Collection is empty collection.

**Unfinished Stubbing Exceptions**

Stubbing consecutive calls with same argument

Stubbing void methods

doNothing() is used for stubbing void methods.

**What is Behavior Verification**.

One of the primary benefits of mock object is that once it gets created, it remembers all operations performed on it.

Behavior Verification help us in verifying that certain mock method was called by system under test or not.

Behavior verification is done by “verify()” method.

Usage: verify(bookRepository).save(book). Whenever unit test executes this call it simply let us know that bookRepository save method was called or not.

Missed implementation of Mockito.verifyNoInteractions(), Mockito.inOrder(<Mock Object>), Mockito.atLeast(NumberOfInvocations), Mockito.atMost(numberOfInvocations), Mockito.atMostOnce(), Mockito.atLeastOnce() and few more

**Configuring Mockito to throw Exception**

For throwing exception in Mockito we use Mockito.when().thenThow(Exception.class).  
In conjunction to this we use

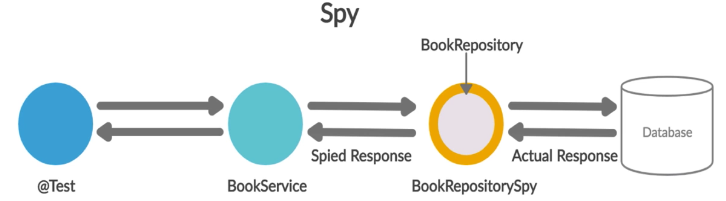
*assertThrows(Exception Class that we are expecting, Action/code which can throw exception)*

for methods that has no return type we use doThrow().when()

**Skipping Argument Capture with and without annotations as it is continued from behavior verification, which I have already skipped.**

**SPY in Mockito**

Spy is a special kind of test double which records information about how they interacted.  
When it comes to Mockito working of spy behaves differently. In addition to recording of information, it also allows us to call normal methods of external dependency. Since external dependency is still being involved, it is very less used. Due to this behavior it is also called as Partial Mock (only in context of mockito)



To spy we use Mockito.spy() method 🡪 It accepts either an object or a class to spy.

We can create spy using @Spy annotation.

Note: When we are performing stubbing using spy, use doReturn().when() instead of when().thenReturn(). When.thenReturn() actually calls the method(Watch Video to understand more).

Behavior Verification in Spy.

**Behavior Driven Development**

It is development process that encourages writing test in a natural, human readable language that focuses on behavior of application.

It defines the style of writing test cases in *“given, when, then”* format.

It also corresponds to “*AAA*” testing i.e. “*Arrange Act Assert*”.

*“Given”* some values and preconditions (*Arrange*)  
“*When*” an action is performed (*Act*)  
“*then*” verify/assert the output (*Assert*)

For Example: “Given” a book, “when” addBook is called, “Then” book is saved to database.

Mockito library contains a class by name BDDMockito, which introduced BDD style API’s.

Demo of Traditional Mockito v/s BDD Mockito

For BDD we use BDDMockito.given().willReturn()

For BDD Assertions we use BDDAssertion class specifically “*then*” method.

*Exception Handling in BDD*

BDDMockito.given(<code that can/will throw exception>).willThrow(Exception.class);

**Argument Matchers in Mockito**

One of the primary benefits of Mockito is ability to perform stubbing of methods and behavior verification.

Mockito uses equals() method to compare the arguments and verify that mock method was called with proper arguments.

Argument Matchers are like wildcard where instead of specific input we can specify range or type of inputs.

For Example:

Mockito.when(bookRepository.findBookById(“1234")).thenReturn(book);  
Instead of using specific string “1234”, we can specify any String using anyString() method.

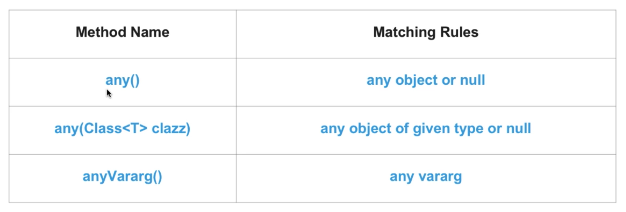
Note: Any type of Argument Matcher is not implemented

Types of Argument Matchers.

1. Type Argument Matchers

**Type Argument Matchers**

They can match to any type of arguments.



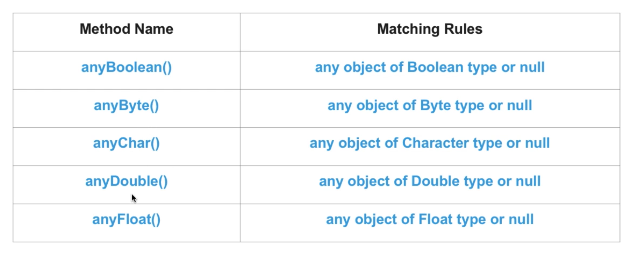
**Invalid use of Argument Matchers.**

Always remember Argument matcher should be provided for all arguments and will probably give InvalidUseOfMatcherException if it is not followed.

Also it is invalid to use argument matchers outside the scope of stubbing or verification and can give PotentialStubbingProblem.

**Specific Type Argument Matchers.**

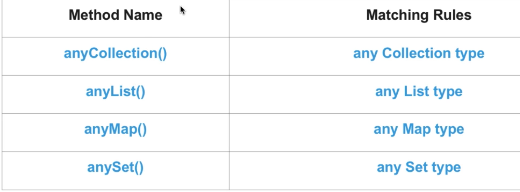
This type of Argument Matchers can match to any object of given type.





**Collection Type Argument Matchers**

This type of argument matchers can match to any collection type based on provided type.



**String Type Argument Matcher**



Mockito Methods

Mockito.mock(class-name) 🡪 The class which we want to mock.

Mockito.when(some content).thenReturn()

Mockito.verify(MockedClass object) 🡪

Mockito.verify(MockedClassObject,Mockito.times(number)) 🡪

Mockito.doNothing() 🡪

Mockito.times() 🡪 oftenly used with verify method. If number of times is 0 then we can also use Mockito .save() method

Mockito.verifyNoInteractions() 🡪

Mockito.verifyNoMoreInteractions() 🡪 It tells us whether no more interactions were held or not.

Mockito.inOrder(<Mock Object>) 🡪 Make sure that a particular sequence is maintained when we call certain methods on mock object.

Mockito.atLeast(NumberOfInvocations) 🡪 It make sures that a particular method is called atleast particular number of times. Failure results in toFewActualInvocations exception. Usually used with verify() method.

Mockito.atMost(numberOfInvocations) 🡪 It makes sure that a particular method is called atMost particular number of times. Usually used with verify() method. Failure results in MoreThanAllowedActualInvocations.

Mockito.atMostOnce() 🡪 It makes sure that a particular method is called at most once. Usually used with verify() method.

Mockito.atLeastOnce() 🡪 It makes sures that a particular method is called at least once. Usually used with verify() method.